

NEW TREATMENT OF THE PULSAR EQUATION

A Thesis

Presented to the Faculty of the Graduate School
of Cornell University

in Partial Fulfillment of the Requirements for the Degree of
Master of Science

by

Michael Joseph Vidal

August 2014

© 2014 Michael Joseph Vidal
ALL RIGHTS RESERVED

ABSTRACT

In solving the pulsar equation, two methods have risen to the forefront, the CKF method (Contopoulos, Kazanas, and Fendt), and the TOTS method (Takamori, Okawa, Takamoto, and Suwa). Both methods are implemented by numerical relaxation, which creates problems at a singular surface known as the light cylinder. Furthermore, these methods give limited information about the problem. The CKF method will not tell you how singular an answer is, only what it will look like after iterative correction. The TOTS method, which foregoes iterative correction, has the potential to give more information, but it has only been tested once, and an extra physical quantity was unnecessarily restricted just to get the solution to converge.

We have replaced relaxation with Newton's method in the context of solving nonlinear equations. This technique is demonstrated by replicating the results of Michel 1973, Contopoulos et al. 1999, Takamori et al. 2012, Lovelace et al. 2006, and Contopoulos et al. 2014. These altered methods refine the original ideas and make clear exactly what place they have in searching for solutions.

We also introduce new investigative paths. We show how we can weed out solutions by revealing the singular behavior of a derivative, even if the function itself appears well-behaved. We also show how we can use a singular solution to generate a smooth solution by looking for smooth contour lines in a field of singular ones with the "lone contour method."

BIOGRAPHICAL SKETCH

Prior to university, Michael Joseph Vidal has always been a hobbyist programmer, independently learning coding through self-inspired pursuits, ranging from a Nintendo Game Boy emulator, to a Japanese Mosaic Puzzle solver, to a file archival experiment. His most recent project is the published “Physics Tutor,” a mobile tutoring app for introductory physics problems available on Android.

In 2008, Michael entered Cornell University, where he would earn his Bachelor of Science. An Engineering Physics major, he has benefited from a diverse repertoire of study, with three undergraduate minors and coursework ranging from the Department of Mathematics to the College of Veterinary Medicine. While earning his bachelor’s degree, Michael performed supersolid research under Professor John D. Reppy, where he assisted with the machining of a liquid helium torsional oscillator, along with data acquisition and analysis. Michael also served as a tutor for the College of Engineering, where he helped his peers through one-on-one instruction.

In 2012, Michael entered Cornell University’s Applied Physics graduate program, where he began his research with Professor Richard V. E. Lovelace. Combining undergraduate and graduate programming instruction, Michael explored a new computer simulation method which enabled further progress in the investigation of neutron star pulsars.

Dedicated to my loving family.

Dedicated to Kimberly Jeanne Beccia and Jennifer Alexis Davis.

Dedicated to my professor and advisor, Richard V. E. Lovelace.

ACKNOWLEDGEMENTS

I would like to acknowledge Earl J. Kirkland for his programming advice and instruction which helped make this research possible.

I would like to acknowledge the School of Applied and Engineering Physics, the Department of Materials Science and Engineering, and the Department of Mathematics for exemplary education and support.

TABLE OF CONTENTS

Biographical Sketch	iii
Dedication	iv
Acknowledgements	v
Table of Contents	vi
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Previous Methods	6
2.1 CKF Method	6
2.2 TOTS Method	7
3 Altered Methods	8
3.1 Adapting Newton's Method	8
3.2 Altered CKF Method	13
3.3 Altered TOTS Method	15
4 Demonstration of Prior Results	16
4.1 Monopole	19
4.2 CKF	19
4.3 TOTS	22
4.4 Jets	25
4.5 Null Sheet	30
5 Additional Critique on TOTS Solutions	32
6 Additional Critique on Jets Solutions	36
7 Conclusions	40
A Boundary Conditions	41
B Smoothing Over The Light Cylinder	44
C Future Opportunities	49
C.1 The Lone Contour Method	49
C.2 Unique Solution?	52
D Mathematical Proof	53
E $H(\psi)$ Dictionary	55

F	Source Code	58
F.1	Main Code	58
F.1.1	matrix.c	59
F.1.2	matrix_build.c	70
F.1.3	inverse.c	73
F.1.4	equation_solver.c	76
F.1.5	equation_solver_2case.c	85
F.1.6	reset_ckf.c	91
F.1.7	reset_tak.c	94
F.1.8	read_matrix.c	96
F.2	Case Files	97
F.2.1	ckf_monopole.c	98
F.2.2	ckf.c	105
F.2.3	ckf_jets.c	114
F.2.4	ckf_null.c	124
F.2.5	tak_monopole_test.c	133
F.2.6	tak_monopole.c	138
F.2.7	tak_monopole_jets.c	143
F.2.8	tak.c	148
F.2.9	tak_jets.c	154
F.2.10	tak_theory_jets.c	160
F.2.11	tak_null.c	165
F.2.12	ckf_monopole_jets.c	171
F.2.13	ckf_tak.c	179
F.2.14	ckf_null_tak.c	188
F.2.15	tak_ckf.c	197
F.2.16	tak_ckf_jets.c	207
F.2.17	tak_ckf_null.c	217
F.2.18	tak_theory_jets_ckf.c	227
	Bibliography	237

LIST OF FIGURES

4.1	The Michel Monopole with $\psi_{eq} = 1$ found using both altered methods (CKF on left, TOTS on right).	19
4.2	Direct replica of the result of [2] (obtained by setting $\psi_{eq} = 1.28$) which can be compared directly to their Figure 3.	20
4.3	CKF solutions found with the Altered CKF method. $\psi_{eq} = 0.001, 0.01, 0.1, 0.5, 0.8, 1.2, 1.4, 1.8, 2.4, 4.0, 40.0, 5000.0$. . .	21
4.4	Direct replica of the result of [3] ($r=0.5$) with the same $\psi_{eq} = 1.225$ which can be compared directly to their Figure 3.	23
4.5	TOTS solutions found with the Altered TOTS method. $\psi_{eq} = 0.001, 0.01, 0.1, 0.5, 0.8, 1.2, 1.4, 1.8, 2.4, 4.0, 40.0, 5000.0$. . .	24
4.6	Jets solutions with $k_H = 5.48$ and $\psi_{eq} = 1$. Left is a modified monopole, right is a modified CKF-like solution.	26
4.7	Monopole jets solution with $k_H = 8.48$ and $\psi_{eq} = 1$. The marked data points show very little change, indicating an “exclusion zone.”	27
4.8	Jets solutions with $k_H = 0, 2$, and 5.48 and $\psi_{eq} = 1$	27
4.9	Theoretical idea for a jets solution.	28
4.10	Null Sheet solution with $\psi_{eq} = 2.0$ using the Altered CKF method.	31
4.11	Null Sheet solution with $\psi_{eq} = 2.0$ using the Altered TOTS method.	31
5.1	TOTS solutions with $\psi_{eq} = 1.0$ using the Altered TOTS method. The grid spacing in both directions is equal, and is 0.05, 0.025, and 0.0125. The second derivative vs. R is plotted next to the corresponding solution and is taken at $Z = 1$	33
6.1	β vs. ψ , with $\psi_{eq} = 2$. From bottom to top, $k_H = 1.7$ (magenta), 1.9 (blue), 2.0 (black), 2.1 (green), 3.0 (red), 4.0 (cyan), 5.0 (magenta), and 7.0 (blue). $k_H = 2$ is the monopole solution ($\beta = 1$). The left plot is data estimated from the lone contour method, and the right plot is of Equation 6.3. See Table 6.1 for the complete list of data.	37
B.1	Various solutions with the Altered CKF method. Top row is monopole ($\psi_{eq} = 1$), then CKF ($\psi_{eq} = 1.28$), then Null Sheet ($\psi_{eq} = 2$). Left column is $Q = -1$, then $Q = 0$, then $Q = 1$	47
B.2	Various solutions with the Altered TOTS method. Top row is monopole ($\psi_{eq} = 1$), then TOTS ($\psi_{eq} = 1.28$), then Null Sheet ($\psi_{eq} = 2$). Left column is $Q = -1$, then $Q = 0$, then $Q = 1$	48
C.1	Jets simulation ($Q = -1, \psi_{eq} = 2, k_H = 3, \beta = 1.3$), demonstrating a lone contour. The entire light cylinder is singular except for one location. Our grid spacing is 0.0125 in both directions, but a higher resolution is recommended for further study.	51

LIST OF TABLES

- 6.1 Data obtained from the lone contour method. k_H and β are fixed in the program, while ψ is found from running the simulation. . . . 39

CHAPTER 1

INTRODUCTION

In the study of rotating neutron stars, or pulsars, there is much interest in the electromagnetic fields and particle wind that emanate from the surface. One key result is the Grad-Shafranov equation of [1], which links the star's rotation to the magnetic flux generated:

$$\left[1 - \left(\frac{r\Omega_*(\psi)}{c}\right)^2\right] \Delta^* \psi - \frac{2r\Omega_*(\psi)^2}{c^2} \frac{\partial \psi}{\partial r} = -\tilde{F}(\psi) \quad (1.1)$$

Where:

$$\begin{aligned} \tilde{F}(\psi) &= \tilde{H}(\psi) \frac{d\tilde{H}(\psi)}{d\psi} \\ \Delta^* &= \partial^2 / \partial r^2 - (1/r)(\partial / \partial r) + \partial^2 / \partial z^2 \end{aligned}$$

$\psi \equiv \psi(r, z)$ is the nonnegative magnetic flux, $\Omega_*(\psi)$ is the angular velocity of the star, c is the speed of light, and $\tilde{H}(\psi)$ is proportional to the current in the poloidal direction. The coordinates are cylindrical, except that by symmetry the angle is removed, leaving r and z . Solving for ψ would in turn allow us to calculate the electric and magnetic fields.

The coefficients of the highest derivative terms simultaneously vanish when $r = |c/\Omega_*(\psi)|$, a location known as the light cylinder. It is well known that many “solutions” returned by simulations have kinks or are otherwise singular at this location. To study this peculiarity, this equation has been simplified further. First, the angular velocity is assumed to be a constant ($\Omega_*(\psi) \longrightarrow \Omega_*$). Then, the

following substitutions are used:

$$\begin{aligned} R &= \left(\frac{\Omega_*}{c} \right) r \\ Z &= \left(\frac{\Omega_*}{c} \right) z \\ H(\psi) &= \left(\frac{\Omega_*}{c} \right) \tilde{H}(\psi) \\ F(\psi) &= \left(\frac{\Omega_*}{c} \right)^2 \tilde{F}(\psi) \end{aligned}$$

Along with the definition of a linear operator:

$$\hat{L} \equiv (1 - R^2) \left(\frac{\partial^2}{\partial R^2} + \frac{\partial^2}{\partial Z^2} \right) - \left(\frac{1 + R^2}{R} \right) \frac{\partial}{\partial R}$$

The result is the so-called pulsar equation:

$$\hat{L}(\psi) = -F(\psi) \tag{1.2}$$

Where the light cylinder now occurs at $R = 1$.

Note that a solution which does exist everywhere will obey the following “smoothness condition” at $R = 1$:

$$\frac{\partial \psi}{\partial R} = \frac{1}{2} F(\psi) \tag{1.3}$$

This is not really a separate piece of information for a true solution, because it is just the pulsar equation with $R = 1$ plugged in. However, many authors single this point out because it is a useful criteria for weeding out potential solutions. Note that in theory, the smoothness condition is enough as is. However, we will show a case where the first derivative seems to behave, and the second derivative is instead used to rule out the solution. Indeed, using the smoothness condition, we show that often, if the solution exists at the light cylinder at all,

then many derivatives with respect to R will exist on that surface too, not just the first (see Appendix D).

We also need to specify boundary conditions. For our purposes, the domain is the set of points where R and Z are both nonnegative. The first boundary is the positive Z -axis, where the value of ψ is agreed upon as zero. Boundary conditions for other edges are not always agreed upon, so throughout this paper various choices will be demonstrated.

Another feature of the pulsar equation is $F(\psi)$, which is a function of the unspecified $H(\psi)$. It is agreed upon that $H(0) = 0$, and it is often proposed that there exists some positive value ψ_{eq} (also called ψ_{open} by some authors) such that $H(\psi) = 0$ if $\psi \geq \psi_{eq}$. In most cases this produces closed loop contours. However, for $0 < \psi < \psi_{eq}$ we will in general have a nonlinear differential equation with an unspecified right-hand side.

Also, in looking at the pulsar equation, say that a solution ψ exists, so that:

$$\hat{L}(\psi) = -F(\psi) \tag{1.4}$$

We desire that for any constant K , $K\psi$ is also a solution. But this means:

$$-F(K\psi) = \hat{L}(K\psi) = K\hat{L}(\psi) = -KF(\psi) \tag{1.5}$$

This forces $F(K\psi) = KF(\psi)$. Seeing as how $F(\psi)$ can be nonlinear, this is not a trivial requirement. Most authors consider cases where this constant is just ψ_{eq} itself and thus accomplish these requirements by fixing:

$$H(\psi) = \psi f\left(\frac{\psi}{\psi_{eq}}\right) \tag{1.6}$$

$$F(\psi) = \psi f\left(\frac{\psi}{\psi_{eq}}\right) \left(f\left(\frac{\psi}{\psi_{eq}}\right) + \frac{\psi}{\psi_{eq}} f'\left(\frac{\psi}{\psi_{eq}}\right) \right) \tag{1.7}$$

Where f is a differentiable function. Note that this form of $H(\psi)$ is sufficient, but not necessary to fulfill the requirement in $F(\psi)$'s behavior.¹

Even with a fixed boundary, this leaves room for debate as to which possibilities for $F(\psi)$ have physical solutions. One example of a debate is whether or not there exists a physical solution without an equatorial current sheet for a given boundary choice. This can be seen by looking at the behavior of $H(\psi)$:

$$\begin{aligned}\lim_{\psi \rightarrow \psi_{eq}^-} H(\psi) &= 0 \longleftrightarrow \text{“There is no equatorial current sheet.”} \\ \lim_{\psi \rightarrow \psi_{eq}^-} H(\psi) &\neq 0 \longleftrightarrow \text{“There exists an equatorial current sheet.”}\end{aligned}$$

Since $H(\psi) = 0$ for $\psi \geq \psi_{eq}$, the current sheet would indicate a discontinuous jump in $H(\psi)$.²

Moreover, as we will see, how to specify $F(\psi)$ is the *only* essential difference between existing numerical methods, and (along with the boundary) is the determinant of whether or not a smooth solution exists. So while ψ is the quantity we ultimately want, $F(\psi)$ is equally as important to consider.

As far as solving the equation itself, no matter how $F(\psi)$ is specified, if we restrict ourselves to $0 \leq R < 1$, then numerical relaxation is a viable method. However, doing this method “as is” will not be able to handle potential singularities at $R = 1$. There have been two alterations to deal with this problem. [2] splits the domain in the CKF method, whereas [3] splits the pulsar equation in the TOTS method. Both methods still use relaxation.

¹We point out that knowing $H(\psi)$ defines $F(\psi)$ uniquely, and because we take $H(\psi)$ to be a nonpositive quantity, knowing $F(\psi)$ also defines $H(\psi)$ uniquely:

$$H(\psi) = -\sqrt{2 \int_0^\psi F(\psi') d\psi'}$$

²If there is no equatorial current sheet, then $F(\psi_{eq}) = H(\psi_{eq}) = 0$. However, as pointed out in [2], if there is an equatorial current sheet, then $F(\psi_{eq})$ is a Dirac delta function and $H(\psi_{eq})$ is undefined. For our purposes, we can just say $F(\psi_{eq}) = H(\psi_{eq}) = 0$ either way.

In this paper we propose a new option: switching relaxation with Newton’s method. While the abstract ideas of the older methods remain, using Newton’s method provides a much more straightforward process conceptually, and eliminates the need for domain splitting, equation splitting, and some extraneous parameters.³

In Chapter 2, we remind the reader of the ideas of the older methods. In Chapter 3, we explain how to adapt Newton’s method to this problem, being mindful of the boundaries and the light cylinder. We define the “Altered CKF method” and “Altered TOTS method” here. In Chapter 4 we demonstrate the altered methods with five previously investigated cases. In Chapter 5 we provide additional comments on the TOTS simulations of [3]. We will see how altering their method makes it much simpler, providing an opportunity to revisit their solutions. This chapter also serves as a warning, that although their method works, one must be cautious about what exactly that means, and what one should and should not expect from an answer. In Chapter 6 we provide additional comments on the jets simulations of [4]. We show how, with the help of the Altered TOTS method, a new investigative tool could possibly help derive functional forms for solutions starting from a singular answer.

³The altered methods do *not* end up merging completely. The treatment of $F(\psi)$ remains a fundamental point of difference.

CHAPTER 2

PREVIOUS METHODS

We give a very simplified summary of the two existing methods. For specific details about the methods, refer to [2] and [3].

2.1 CKF Method

1. Initialize ψ and $F(\psi)$ to values that favor numerical convergence, and split the domain into three regions: inside, outside, and on the light cylinder.¹
2. Apply an iteration of numerical relaxation on the pulsar equation both inside and outside the light cylinder separately. The light cylinder serves as a boundary edge for both regions.
3. Correct the distribution of $F(\psi)$ as follows:
 - a) Determine $F(\psi)$ on the light cylinder by the smoothness condition (Equation 1.3). Also set ψ on the light cylinder as the average of the values to the immediate left and right.
 - b) For points elsewhere, because $F(\psi)$ only depends on ψ , $F(\psi)$ must be the same value everywhere on a field line, *including* the point where it crosses the light cylinder. Consider that on the light cylinder, $F(\psi)$ is known from a), so we can use this knowledge to set $F(\psi)$ everywhere along each field line.
4. Repeat steps 2 and 3 until convergence.

¹Personal observations suggest that the light cylinder should lie exactly on the grid for all simulation methods.

2.2 TOTS Method

1. Split the pulsar equation into two equations:

Ampere's law:

$$\frac{\partial^2 \psi}{\partial R^2} + \frac{\partial^2 \psi}{\partial Z^2} - \frac{1}{R} \frac{\partial \psi}{\partial R} = -ST(R, Z) \quad (2.1)$$

Force-free condition:

$$R^2 \left(\frac{\partial^2 \psi}{\partial R^2} + \frac{\partial^2 \psi}{\partial Z^2} - \frac{1}{R} \frac{\partial \psi}{\partial R} \right) + 2R \frac{\partial \psi}{\partial R} - F(\psi) = -ST(R, Z) \quad (2.2)$$

Note the new unknown term $ST(R, Z)$, which is the toroidal current.

2. Fix a functional form for $H(\psi)$ (and thus $F(\psi)$) and initialize ψ and $ST(R, Z)$ to values that favor numerical convergence (in this method, $F(\psi)$ is known, but ψ and $ST(R, Z)$ are unknown, so we still have two unknowns).

3. Apply an iteration of numerical relaxation on Ampere's law over the entire region (straight through the light cylinder).

4. Update $ST(R, Z)$ values (refer to [3]). Note that this is where the Force-free condition comes into play.

5. Repeat steps 3 and 4 until convergence.

CHAPTER 3

ALTERED METHODS

Both previous methods use different ideas to deal with the light cylinder, and both are implemented with the help of numerical relaxation. We believe that a benefit can be found by replacing their implementation with Newton's method and nonlinear equation solving.

3.1 Adapting Newton's Method

We remind the reader of the steps of Newton's method, with the context of using CKF's and TOTS's ideas. Similar to relaxation, we take the pulsar equation and convert it to a discrete version. For now, we consider $F(\psi) = 0$. We will consider the general case with $F(\psi) \neq 0$ later. The pulsar equation then becomes:

$$a_1 U_{j-1,k} + a_2 U_{j+1,k} + a_3 U_{j,k-1} + a_4 U_{j,k+1} + a_5 U_{j,k} = a_6$$

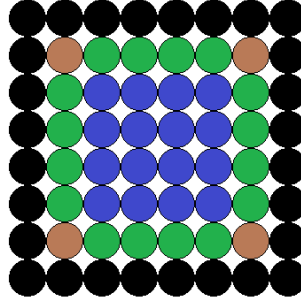
$$\begin{aligned} a_1 &= 1 - j^2(\Delta R)^2 + \frac{1}{j} + j(\Delta R)^2 \\ a_2 &= 1 - j^2(\Delta R)^2 \\ a_3 &= \left(\frac{\Delta R}{\Delta Z}\right)^2 (1 - j^2(\Delta R)^2) \\ a_4 &= \left(\frac{\Delta R}{\Delta Z}\right)^2 (1 - j^2(\Delta R)^2) \\ a_5 &= \left[-2 - 2\left(\frac{\Delta R}{\Delta Z}\right)^2\right] (1 - j^2(\Delta R)^2) - \frac{1}{j} - j(\Delta R)^2 \\ a_6 &= 0 \end{aligned}$$

(3.1)

Where we have chosen to use backwards difference for the first derivative and

central difference for the second derivatives. One is welcome to use different choices, and in particular, because we do not have to solve for $U_{j,k}$ directly in the equation we make, it is *not* required to have a_5 nonzero everywhere, as in relaxation. ΔR and ΔZ are the desired grid spacings. $R = j\Delta R$, $Z = k\Delta Z$, and discretization comes from indexing j and k as nonnegative integers.

Now, to demonstrate how to deal with boundaries, consider the following grid:



Boundary points are black.

For all interior points not touching a boundary (“interior interior” points, labeled blue), we can simply write an equation for that point in terms of itself and the surrounding points using Equation 3.1.

For all interior points adjacent to one boundary (edge points, labeled green), we write one equation for the interior point using Equation 3.1, and one equation for the adjacent boundary point of the form:

$$b_1 U_{j-1,k} + b_2 U_{j+1,k} + b_3 U_{j,k-1} + b_4 U_{j,k+1} + b_5 U_{j,k} = b_6 \quad (3.2)$$

The boundary equation is solved for that boundary point, and plugged into the interior point’s equation. For example, listing the coefficients as a tuple

$(b_1, b_2, b_3, b_4, b_5, b_6)$, consider some typical bottom boundary conditions:

$$\begin{aligned}\psi(R_0, Z_0) &= f_0 \longrightarrow (0, 0, 1, 0, 0, f_0) \\ \frac{\partial \psi}{\partial Z}|_{(R_0, Z_0)} &= f_0 \longrightarrow (0, 0, -1, 0, 1, f_0 \Delta Z)\end{aligned}$$

Where f_0 is a constant, and (R_0, Z_0) is a point on the bottom boundary.

For all interior points touching two boundaries (corner points, labeled brown), both boundary points are eliminated by boundary-derived equations.¹

Thus far we have not made any accommodation for the light cylinder (other than placing the light cylinder exactly on the grid). But it is generally desired that the solution be smooth over the light cylinder. If N_{LC} is the grid number of the light cylinder, then consider the following overwrite to Equation 3.1:

$$\text{If } N_{LC} - 1 \leq j \leq N_{LC} + 1$$

$$\text{Then } (a_1, a_2, a_3, a_4, a_5, a_6) = (-1/2, -1/2, 0, 0, 1, 0)$$

This makes the solution smooth over the light cylinder.²

Note that for an $N_R \times N_Z$ grid, this amounts to solving $(N_R - 2) \times (N_Z - 2)$ equations for $(N_R - 2) \times (N_Z - 2)$ unknowns. Let $S = (N_R - 2) \times (N_Z - 2)$, and let the variables be given by v_m with $0 \leq m \leq S - 1$. Note that we have a linear system of S equations for S variables. For convenience, let A and B be the coefficient matrix and right hand side vector for that system.

Now, these equations are linear, but we still need to include $F(\psi)$, which in general will not be linear. Reconsidering $F(\psi)$, one way to write each equation

¹See Appendix A for more details about the boundaries.

²See Appendix B for more details about the need (or lack or need) for smoothing over the light cylinder.

is:³

$$eq_m : \sum_{n=0}^{S-1} A_{m,n} v_n - B_m + F_m = 0 \quad (3.3)$$

Note that $F(\psi)$ is a function of ψ only, so for any grid point represented by the variable v_m , we write $F_m \equiv F(v_m)$.

Now, for Newton's method, we need the Jacobian matrix. Fortunately, it is easy to write the Jacobian analytically, with elements:

$$J_{m,n} = \frac{\partial eq_m}{\partial v_n} = A_{m,n} + \delta_{m,n} \frac{dF_m}{dv_n} \quad (3.4)$$

In the CKF method, the derivative of F_m can be found by the chain rule, whereas in the TOTS method, the derivative of F_m can be found using the guessed formula of $F(\psi)$. We will see later that this term ends up being unimportant. For now, just note that the off-diagonal elements of J are fixed. Only the diagonal ones ever change.

One straightforward (but by no means the most efficient) way to perform one iteration of Newton's method is as follows:

³Even if $F(\psi)$ is linear or has a linear term, here we keep $F(\psi)$ entirely separate.

1. Update diagonal elements of J .
2. Update J inverse (J^{-1}).
3. Update all eq_m .
4. For each variable v_m , perform the following algorithm:

Input: v_{old} , the existing value of that variable.

Output: v_{new} , the new value of that variable.

$$\begin{aligned}
 t &:= \sum_{n=0}^{S-1} J_{m,n}^{-1} eq_n \\
 t &:= v_{old} - t \\
 v_{new} &:= (FRAC)(t) + (1 - FRAC)(v_{old})
 \end{aligned}$$

Where FRAC is similar to the relaxation parameter. One key advantage we have noticed is that FRAC follows a very simple rule: Higher FRAC converges faster, and lower FRAC converges more carefully. The relaxation parameter, on the other hand, only vaguely followed that rule. There would often be exceptions, making it harder to work with.

Now, we take advantage of a major shortcut which makes the speed of Newton's method competitive with relaxation. Our observation is that the updates to J and J^{-1} do not play an important role. Thus, we can fix $J = A$, $J^{-1} = A^{-1}$ once. Also note that A only depends on the grid size, grid spacing, and choice of boundary conditions. Working with a fixed grid and boundary, one could store J^{-1} in memory, negating the need for steps 1 and 2 entirely.

Now, replacing numerical relaxation with Newton's method is straightforward. Consider these modified processes:

3.2 Altered CKF Method

1. Initialize ψ to 1 and $F(\psi)$ to 0 everywhere.⁴
2. Apply an iteration of Newton's method.
3. Correct the distribution of $F(\psi)$ as in the original CKF method (this step does not change).
4. Repeat steps 2 and 3 until convergence.

In our opinion, splitting the domain was an implementation detail, not a fundamental part of the CKF method. Their fundamental idea was to enforce the smoothness of ψ over the light cylinder, which we have ensured through the equations themselves. Thus, although we pay heed to the light cylinder in crafting the equations, once we perform Newton's method, light cylinder points are like any other.

It should be noted that when we demonstrate this altered method in Chapter 4, we deviate from the spirit of the original CKF method in two ways. First, CKF used a coordinate transformation to accommodate an infinite grid. To avoid this, we use a technique proposed by [4]. This allows us to deal with field lines that would cross the light cylinder, but cannot because the grid ends prematurely. Second, the original CKF method seeks solutions where ψ_{eq} is found iteratively, by solving inside the light cylinder and setting ψ_{eq} to whatever value is found in the lower right corner of the inside region for that iteration. However, we want to draw direct comparisons between CKF-like solutions and other so-

⁴This particular choice for initialization was chosen for convenience. The only real requirement is that the method converge to a solution where ψ is positive everywhere, making this a sensible choice.

lutions which take ψ_{eq} as an adjustable parameter, so we forego this requirement here and also take ψ_{eq} as an adjustable parameter.

If desired, one can work with CKF's original requirements. The coordinate transformation would simply amount to using a transformed set of equations and boundary conditions which are already laid out in [2]. As for ψ_{eq} , one could simply modify the equations for any affected boundary points to be equal to the appropriate corner point (this boundary does not strictly fit the form of the boundary equations we laid out prior, but the idea is still easy to implement. See Appendix A).

3.3 Altered TOTS Method

1. Fix a functional form for $H(\psi)$ (and thus $F(\psi)$).
2. Initialize ψ to 1 everywhere.
3. Keep applying iterations of Newton's method until convergence.

The TOTS method becomes much more direct. There is no need to split the pulsar equation or introduce the toroidal current as is done in [3]. Our belief is that these were again simply implementation details, and not fundamental parts of their method.

Also note that with domain splitting and toroidal currents aside, both CKF's and TOTS's ideas are more similar than one would believe looking at the steps of their original methods. The one fundamental difference in how they approach the problem is evident: one iterates to find $F(\psi)$, one sets $F(\psi)$ directly.

CHAPTER 4

DEMONSTRATION OF PRIOR RESULTS

To prove the usefulness of these altered methods, we will demonstrate the replications of five previously found results (Monopole, CKF, TOTS, Jets, and Null Sheet).

It should be noted that we use the following boundary conditions:

Left Edge:

$$\psi(R_0, Z_0) = 0$$

Top and Right Edge:

$$R_0 \frac{\partial \psi}{\partial R} \Big|_{(R_0, Z_0)} + Z_0 \frac{\partial \psi}{\partial Z} \Big|_{(R_0, Z_0)} = 0$$

Bottom Edge:

For Monopole:

$$\psi(R_0, Z_0) = \psi_{eq}$$

For Null Sheet:

$$\begin{aligned} \frac{\partial \psi}{\partial Z} \Big|_{(R_0, Z_0)} &= 0 & R_0 &\leq 1 \\ \frac{\partial \psi}{\partial Z} \Big|_{(R_0, Z_0)} &= \frac{H(\psi(R_0, Z_0))}{(R_0^2 - 1)^{1/2}} & R_0 &> 1 \end{aligned}$$

All other cases:

$$\begin{aligned} \frac{\partial \psi}{\partial Z} \Big|_{(R_0, Z_0)} &= 0 & R_0 &\leq 1 \\ \psi(R_0, Z_0) &= \psi_{eq} & R_0 &> 1 \end{aligned}$$

(4.1)

Where ψ_{eq} is specified directly. $F(0) = 0$ and $F(\psi) = 0$ if $\psi \geq \psi_{eq}$. For $0 < \psi < \psi_{eq}$, the Altered CKF method finds $F(\psi)$ iteratively, but when using the Altered TOTS method, we specify $F(\psi)$:

Monopole:

$$\begin{aligned} H(\psi) &= \psi \left(\frac{\psi}{\psi_{eq}} - 2 \right) \\ F(\psi) &= 2\psi \left(\frac{\psi}{\psi_{eq}} - 1 \right) \left(\frac{\psi}{\psi_{eq}} - 2 \right) \end{aligned}$$

TOTS:

$$\begin{aligned} H(\psi) &= \psi \left(\frac{\psi}{\psi_{eq}} - 1 \right) \\ F(\psi) &= 2\psi \left(\frac{\psi}{\psi_{eq}} - 1 \right) \left(\frac{\psi}{\psi_{eq}} - \frac{1}{2} \right) \end{aligned}$$

Jets:

$$\begin{aligned} H(\psi) &= \frac{k_H}{2} \psi \left(\beta \frac{\psi}{\psi_{eq}} - 2 \right) \\ F(\psi) &= \frac{k_H^2}{2} \psi \left(\beta \frac{\psi}{\psi_{eq}} - 1 \right) \left(\beta \frac{\psi}{\psi_{eq}} - 2 \right) \end{aligned}$$

Null Sheet:

$$\begin{aligned} H(\psi) &= 1.07\psi \left(2 - \frac{\psi}{\psi_{eq}} \right) \left(1 - \frac{\psi}{\psi_{eq}} \right)^{0.4} \\ F(\psi) &= \frac{2}{5} (1.07)^2 \psi \left[6 \left(\frac{\psi}{\psi_{eq}} \right)^2 - 12 \frac{\psi}{\psi_{eq}} + 5 \right] \left(2 - \frac{\psi}{\psi_{eq}} \right) \left(1 - \frac{\psi}{\psi_{eq}} \right)^{-0.2} \end{aligned} \tag{4.2}$$

Note that we are not always using the outer boundary conditions originally considered by the authors. However, it is generally believed that the particular boundary conditions far away do not matter, and the ones for the left and bottom edge are agreed upon. For cases with a star, it is inserted as a 2x2 box in the lower left corner:

Jets:

$$\psi_{star} = \frac{1}{(R^2 + Z^2)^{\frac{1}{2}}}$$

All other cases:

$$\psi_{star} = \frac{R^2}{(R^2 + Z^2)^{\frac{3}{2}}} \tag{4.3}$$

Also, all simulations in this section are done with a grid spacing of 0.05 in both directions, and most simulations use a 40x40 grid (so they cover $0 \leq R, Z \leq 2$).¹

¹Note that it is not hard to enter in different boundary conditions if desired. Also everything said here assumes a domain with only nonnegative Z . These decisions need to be altered if one wanted to use all points in the closed half-plane $R \geq 0$ in a simulation.

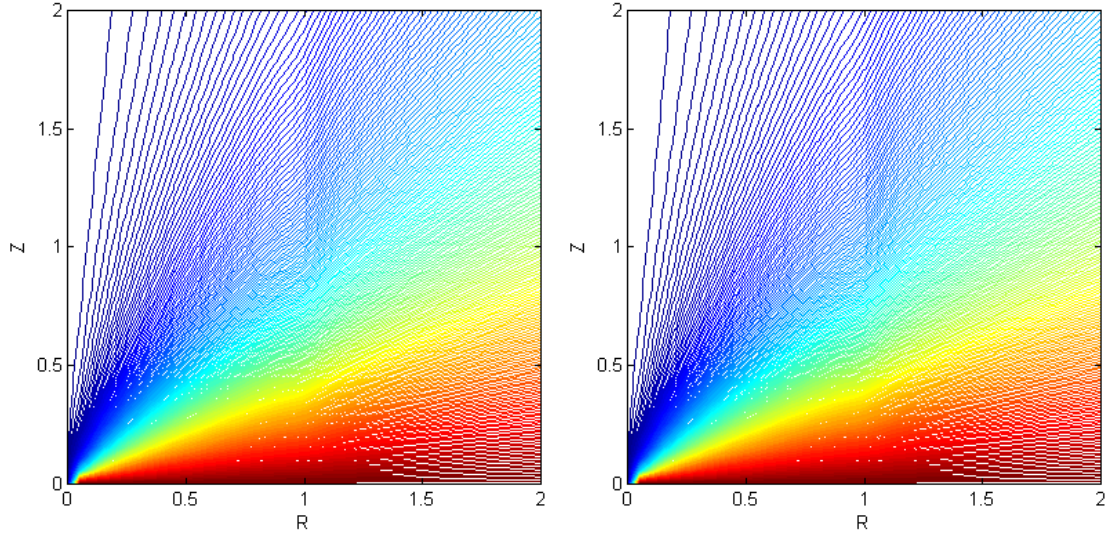


Figure 4.1: The Michel Monopole with $\psi_{eq} = 1$ found using both altered methods (CKF on left, TOTS on right).

4.1 Monopole

From [5], the monopole solution is reproduced with both altered methods in Figure 4.1.

4.2 CKF

From [2], and as seen in [4] and [6] (not an exhaustive list), reproduced using Altered CKF.

Figure 4.2 shows a more detailed version of a case in [2]. Figure 4.3 shows a variety of CKF solutions. There are three regions of ψ_{eq} of interest. Low ψ_{eq} has solutions mostly featuring closed contour loops, intermediate ψ_{eq} has “typical”

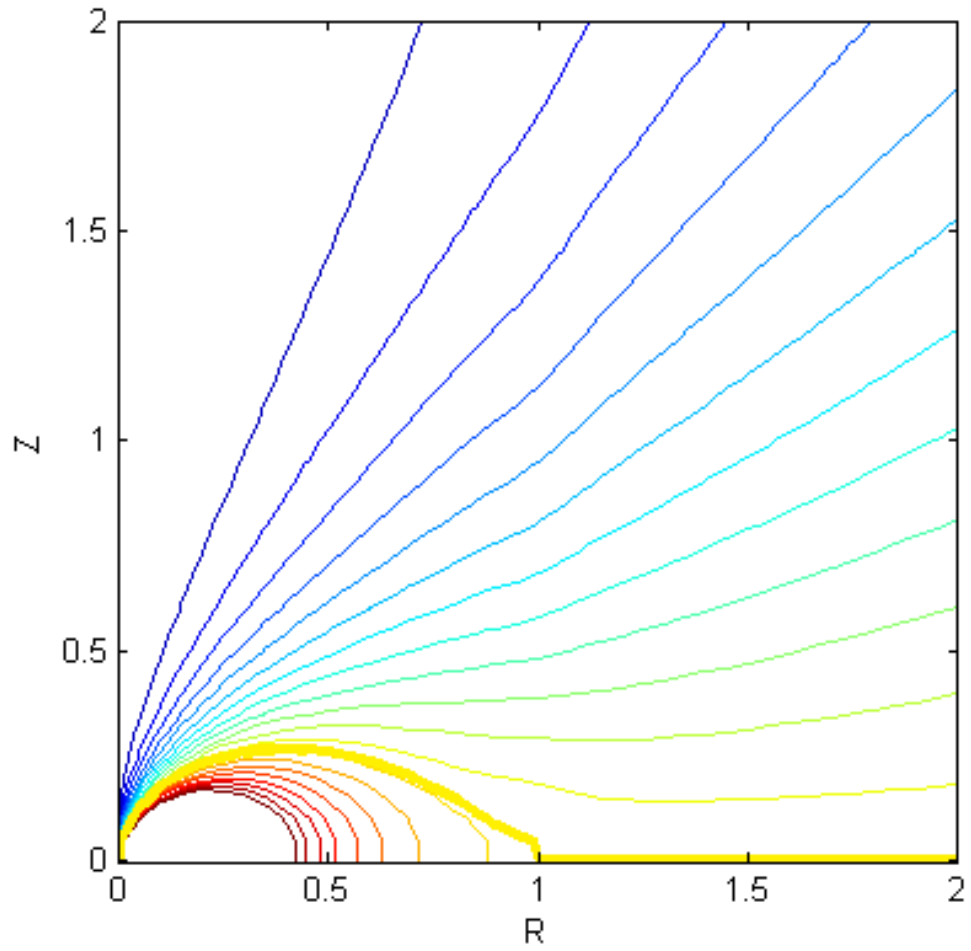


Figure 4.2: Direct replica of the result of [2] (obtained by setting $\psi_{eq} = 1.28$) which can be compared directly to their Figure 3.

CKF solutions, and high ψ_{eq} shows an introduction of field lines which do not emanate from the star at all.

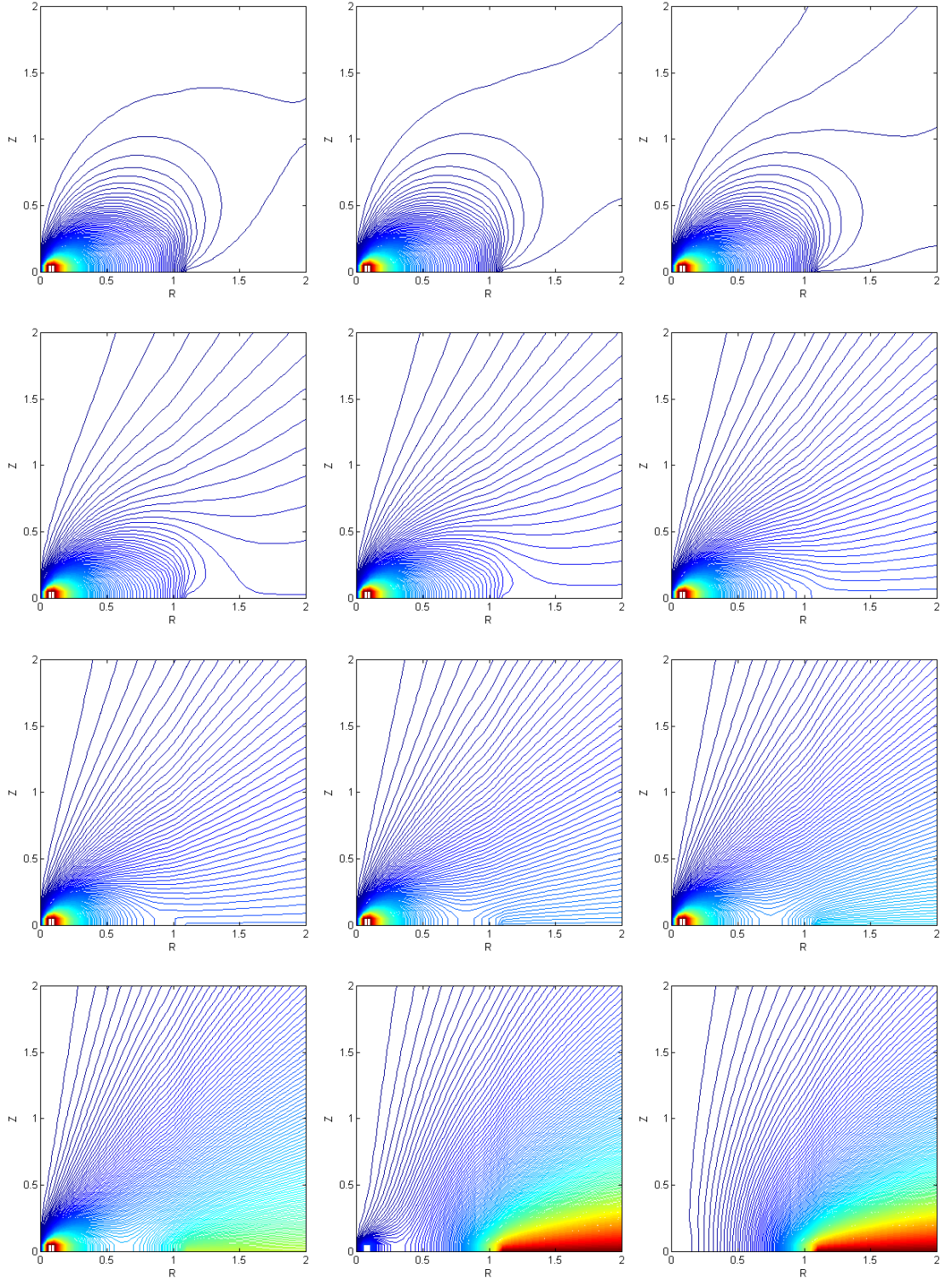


Figure 4.3: CKF solutions found with the Altered CKF method.
 $\psi_{eq} = 0.001, 0.01, 0.1, 0.5, 0.8, 1.2, 1.4, 1.8, 2.4, 4.0, 40.0, 5000.0$

4.3 TOTS

From [3]. As previously stated, the Altered TOTS method does not need to introduce the toroidal current, so some parameters mentioned in their original paper are unnecessary.

Figure 4.4 shows a more detailed version of a case in [3]. Figure 4.5 shows a variety of TOTS solutions. From their equations, $F(\psi) = A^2\psi(\psi - \psi_{ret})(\psi - \psi_{eq})$, $A^2 = \frac{1}{r\psi_{eq}^2}$, and we used a fixed $r \equiv \frac{\psi_{ret}}{\psi_{eq}} = 0.5$, although other values $0.5 \leq r \leq 1.0$ they considered could be used without difficulty.

When using the same ψ_{eq} value, the TOTS solution generally looks like the CKF solution, but with a ripple near the light cylinder. In Chapter 5 we provide more analysis on this point.

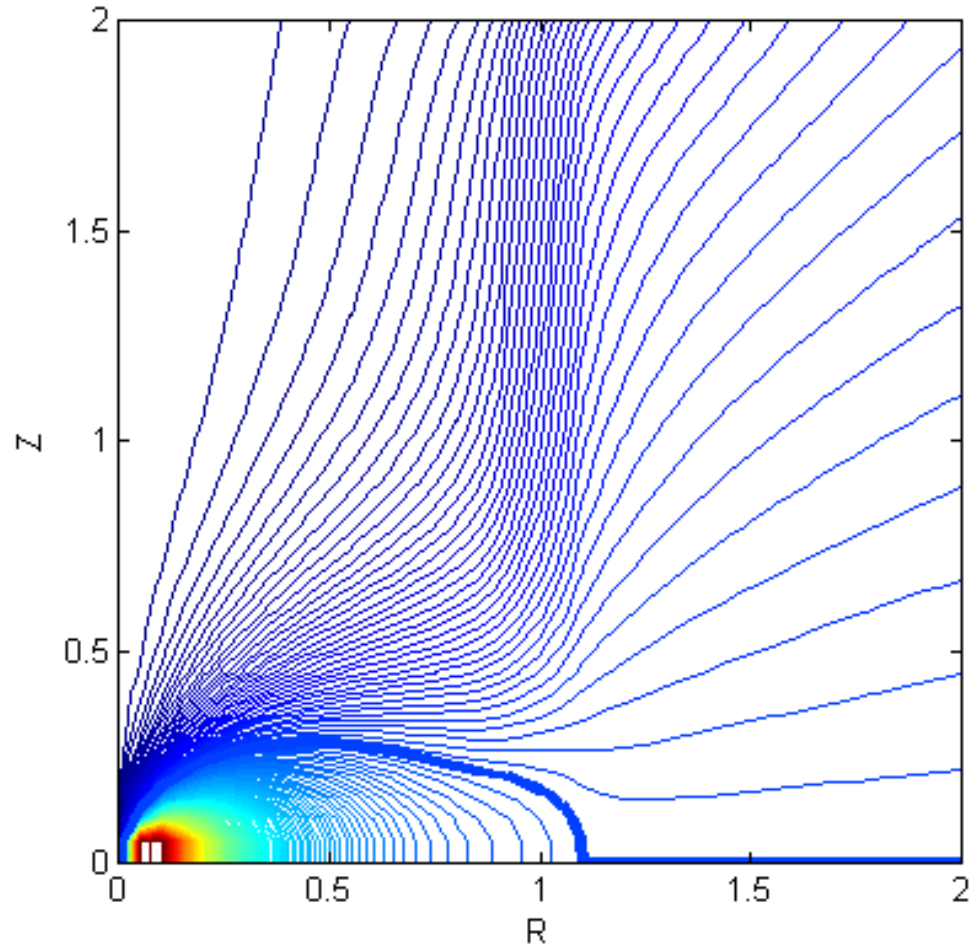


Figure 4.4: Direct replica of the result of [3] ($r=0.5$) with the same $\psi_{eq} = 1.225$ which can be compared directly to their Figure 3.

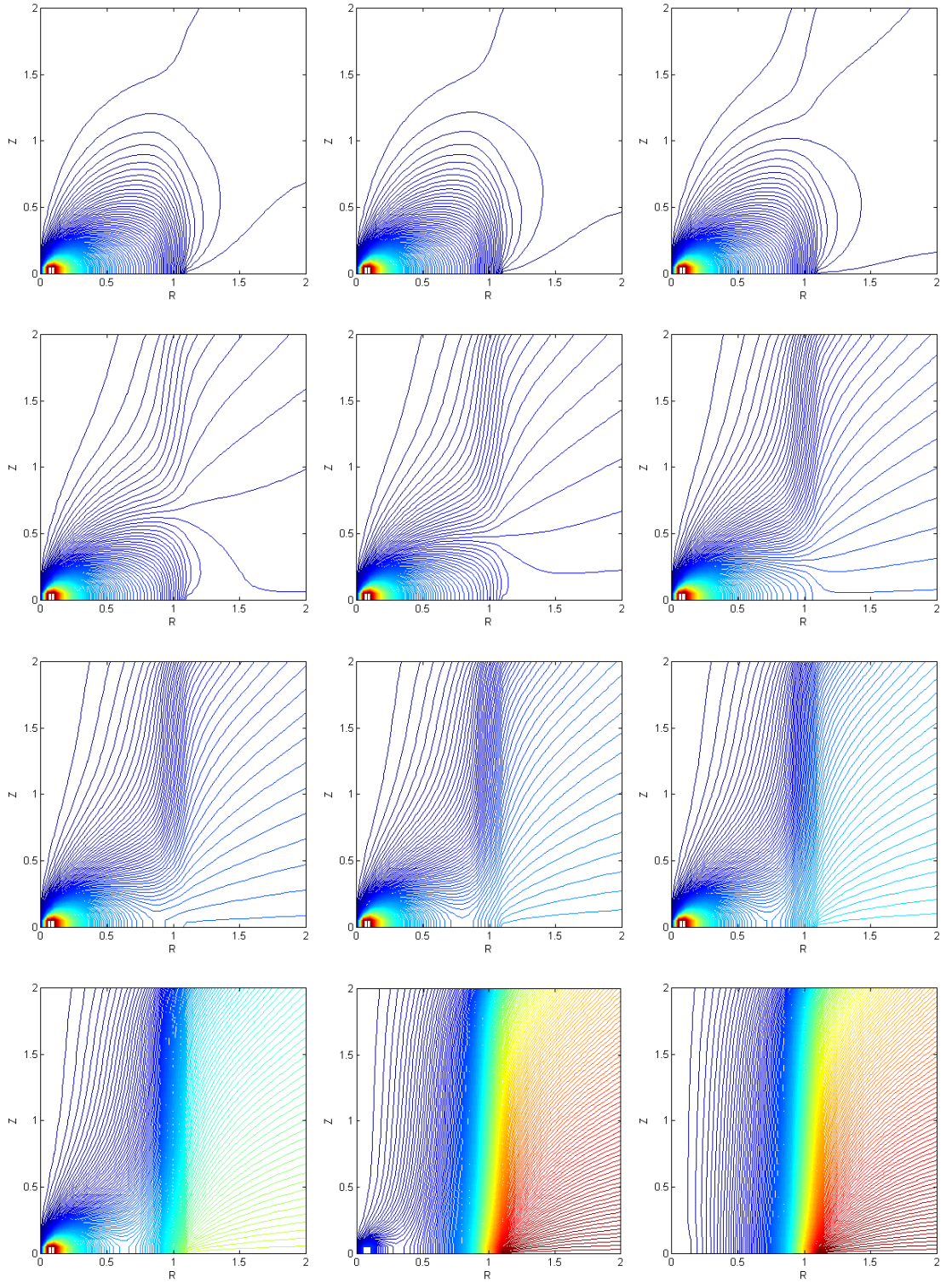


Figure 4.5: TOTS solutions found with the Altered TOTS method.
 $\psi_{eq} = 0.001, 0.01, 0.1, 0.5, 0.8, 1.2, 1.4, 1.8, 2.4, 4.0, 40.0, 5000.0$

4.4 Jets

From [4] is a solution with collimated jets along the Z-axis. They use the CKF method, but for field lines that do not cross the light cylinder, they use:

$$H(\psi) = \frac{k_H}{2} \psi \left(\beta \frac{\psi}{\psi_{eq}} - 2 \right) \quad (4.4)$$

$$F(\psi) = \frac{k_H^2}{2} \psi \left(\beta \frac{\psi}{\psi_{eq}} - 1 \right) \left(\beta \frac{\psi}{\psi_{eq}} - 2 \right) \quad (4.5)$$

Where k_H is an adjustable parameter, and β is determined iteratively using:

$$\beta = \frac{1}{2} \left\{ 3 - \left[1 + \frac{8\psi_c F_c}{k_H^2} \right]^{\frac{1}{2}} \right\} \quad (4.6)$$

ψ_c and F_c are the values of ψ and $F(\psi)$ at the upper right corner of the region inside the light cylinder. We note that β always ended up negligibly different from 1 in these particular simulations.

There was much difficulty in the numerical stability of such solutions when using relaxation. Using the Altered CKF method seems to alleviate these problems. To demonstrate this, in Figure 4.6 we show that one can modify both a monopole solution and a CKF-like solution by adding jets. An interesting feature we point out in Figure 4.7 is an “exclusion zone” where ψ approaches a constant value, consistent with an observation of [4].

If we take $\beta = 1$ for the time being, the form of $H(\psi)$ for jets is a generalization of the monopole $H(\psi)$ with a varying constant in front. Indeed (restricting ourselves inside the light cylinder) it is straightforward to show three types of curvature based on k_H in Figure 4.8. For $0 \leq |k_H| < 2$, jets curve to the right, for $|k_H| = 2$, we get the straight line monopole solution, and $|k_H| > 2$ we get jets that curve upwards. This would suggest that this jets idea could lead to a generaliza-

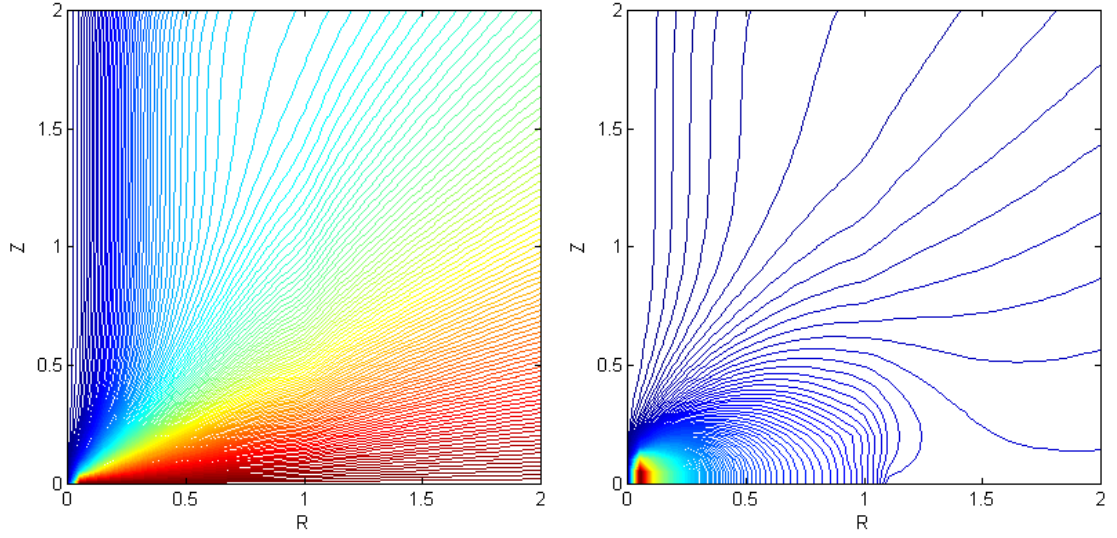


Figure 4.6: Jets solutions with $k_H = 5.48$ and $\psi_{eq} = 1$. Left is a modified monopole, right is a modified CKF-like solution.

tion of the original monopole solution. But of course, the monopole solution is smooth through the light cylinder, whereas contour lines that form a jet (curved in either direction) appear to never lead to a smooth solution. There is one possible loophole for the upward jets: perhaps there exists some ψ^* such that we can set $H(\psi) = \frac{k_H}{2}\psi(\beta\frac{\psi}{\psi_{eq}} - 2)$ for $0 \leq \psi \leq \psi^*$ only and have those field lines never touch the light cylinder at all, avoiding the smoothness requirements.

Whether or not this can be true remains an open question. On the one hand, personal observations suggest that no matter what value of ψ^* one chooses, there is always some grid range that would make that value of ψ^* cross the light cylinder and thus ruin the solution. One cannot use this idea to rule out every nonzero value of ψ^* because this would require an infinite grid. However, with a finite grid, one could rule out any arbitrarily small value of ψ^* , essentially doing the same thing.

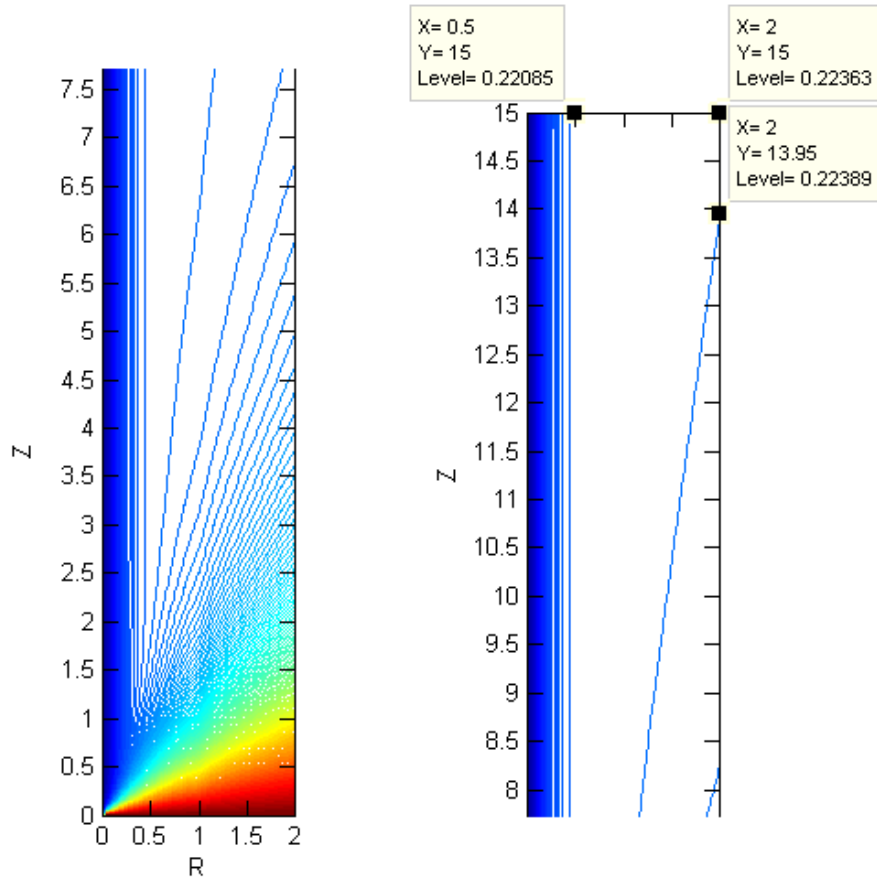


Figure 4.7: Monopole jets solution with $k_H = 8.48$ and $\psi_{eq} = 1$. The marked data points show very little change, indicating an “exclusion zone.”

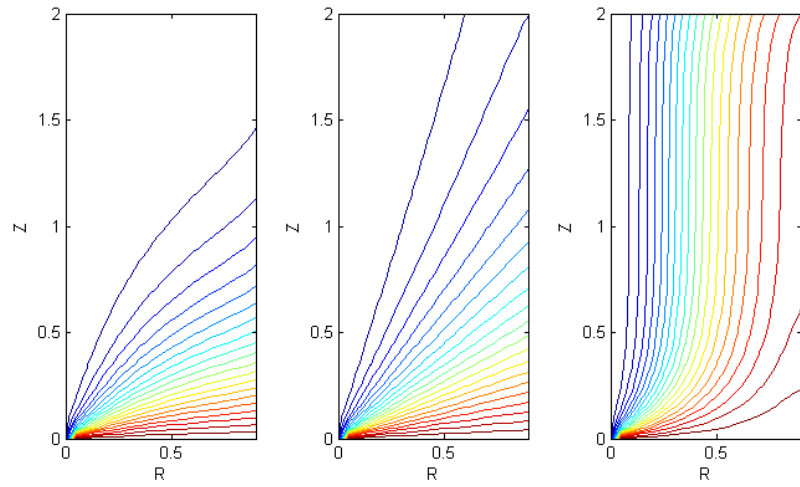


Figure 4.8: Jets solutions with $k_H = 0, 2$, and 5.48 and $\psi_{eq} = 1$.

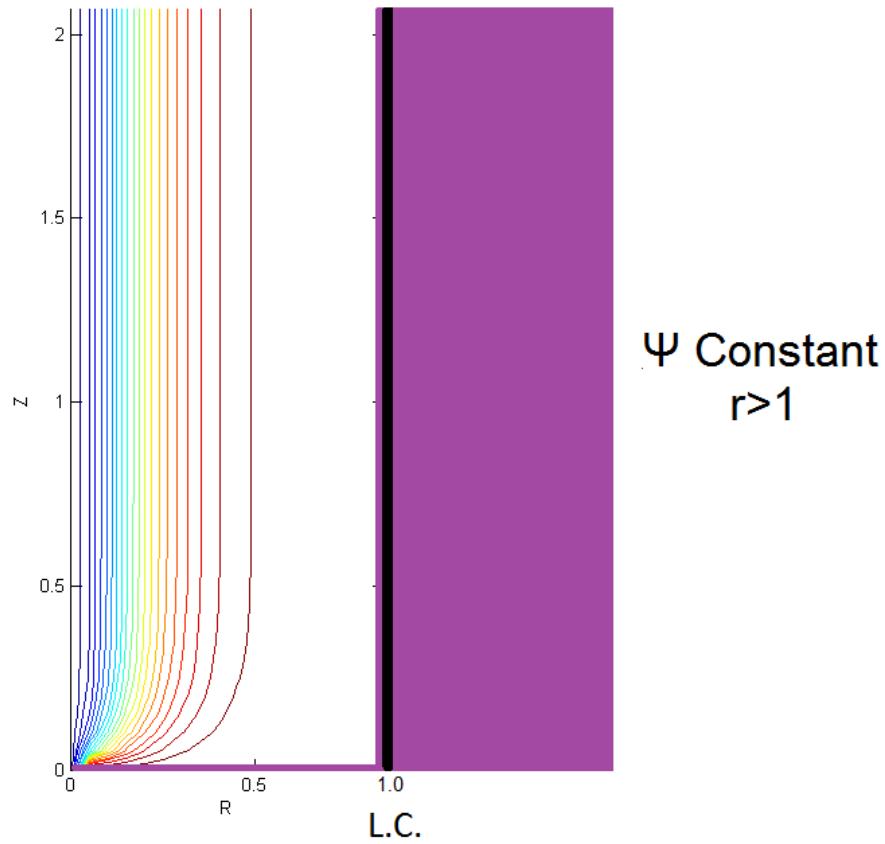


Figure 4.9: Theoretical idea for a jets solution.

On the other hand, imagine the theoretical picture in Figure 4.9. Consider if all the space within the light cylinder was filled with jets emanating from the origin. Further consider that there is one field line that travels from the origin along the equator, and then shoots up the light cylinder. Define this line as ψ_{eq} . Now past the light cylinder ψ is simply the constant value of ψ_{eq} (this is a more drastic example of an “exclusion zone”). Although this solution extends past the light cylinder in a trivial fashion, it does technically satisfy all of the boundary conditions. Further, this solution would be smooth over the light cylinder, and would have no jet line crossing the light cylinder ruining the solution. In terms

of F , we would have:

$$F(\psi) = \frac{k_H^2}{2} \psi \left(\beta \frac{\psi}{\psi_{eq}} - 1 \right) \left(\beta \frac{\psi}{\psi_{eq}} - 2 \right) \quad 0 \leq \psi < \psi_{eq} \quad (4.7)$$

$$F(\psi) = 0 \quad \psi \geq \psi_{eq} \quad (4.8)$$

The two main differences between this theoretical solution and other jets solutions we have studied are that this theoretical solution has the entire region inside the light cylinder filled with jets, and has no jet contour crossing the light cylinder. One or both of these differences could be the key to having a viable jets solution. Of course, whether or not this is an actual solution to the pulsar equation remains to be seen.

One last issue is whether β really is just 1. Perhaps it is not a constant at all, but rather a function of ψ . In Chapter 6 we provide more analysis on this point.

4.5 Null Sheet

A recent case presented in [7] presents an opportunity to demonstrate a more complex boundary condition. This new case was similar to [2], but used a different bottom boundary condition (for R_0 past the light cylinder). We implemented this using:

$$(b_1, b_2, b_3, b_4, b_5) = (0, 0, -1, 0, 1) \quad (4.9)$$

$$b_6 = -\Delta Z \sqrt{\frac{2 \int_0^{\psi(R_0, Z_0)} F(\psi') d\psi'}{j^2(\Delta R)^2 - 1}} \quad (4.10)$$

Where the integral is done using the trapezoidal rule. Unlike other cases, b_6 had to be updated every iteration. Figure 4.10 shows an attempt to replicate the behavior of [7] using the Altered CKF method. Convergence in this case was less favorable, suggesting that a finer and further-reaching grid would be required to study this case correctly. We do note that the behavior of the contour lines is consistent with what [7] saw.

As a further demonstration opportunity, we considered the fitting expression for $H(\psi)$ given in [7], which could then be fed into the Altered TOTS method. After our substitutions:

$$H(\psi) = 1.07\psi \left(2 - \frac{\psi}{\psi_{eq}}\right) \left(1 - \frac{\psi}{\psi_{eq}}\right)^{0.4} \quad (4.11)$$

$$F(\psi) = \frac{2}{5}(1.07)^2\psi \left[6 \left(\frac{\psi}{\psi_{eq}}\right)^2 - 12\frac{\psi}{\psi_{eq}} + 5\right] \left(2 - \frac{\psi}{\psi_{eq}}\right) \left(1 - \frac{\psi}{\psi_{eq}}\right)^{-0.2} \quad (4.12)$$

Figure 4.11 shows the result. Although the graph has a slight light cylinder ripple, the behavior is similar. If nothing else, this suggests that this form for $H(\psi)$ is a reasonable approximation.

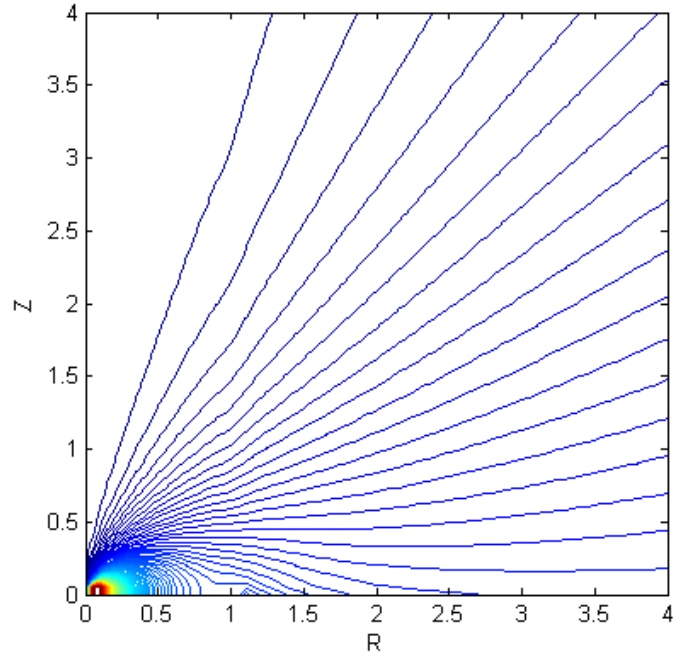


Figure 4.10: Null Sheet solution with $\psi_{eq} = 2.0$ using the Altered CKF method.

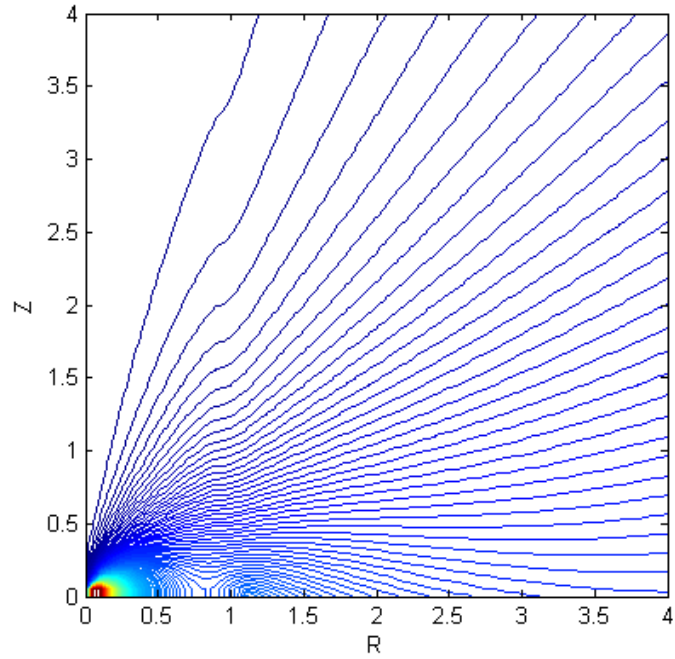


Figure 4.11: Null Sheet solution with $\psi_{eq} = 2.0$ using the Altered TOTS method.

CHAPTER 5

ADDITIONAL CRITIQUE ON TOTS SOLUTIONS

Here we show the cause of the curious ripples at the light cylinder of TOTS solutions. As we have mentioned, the Altered TOTS method is very straightforward conceptually. Make a guess for $F(\psi)$, and then the method will solve for ψ . Indeed, if you use $F(\psi)$ for the monopole, you get the correct monopole answer. Also, after generating a CKF answer, one can take the $F(\psi)$ values produced and plug them into Altered TOTS to generate the exact same CKF answer. Note that in both of these cases, one gets a smooth solution across the light cylinder.

In contrast, although similar to CKF answers, the graphs in Figures 4.4 and 4.5 have ripples at the light cylinder. However, so far it seems like a minor issue. Perhaps this is an artifact of using a coarse grid, or maybe the guess of $F(\psi)$ used simply gives a solution with field lines curving slightly upwards. Remember, this guess is meant to be an approximation, after all.

In a strict mathematical sense, however, we believe that the simulations actually return singular solutions, but that a combination of a coarse grid and unnecessary smoothing have disguised this fact. In Appendix B we will show the affect of taking away smoothing. However, someone doing simulations with smoothing may not see such a drastic effect, and may not even suspect the solution might be singular at all. Here we present another way of seeing singular behavior. In Figure 5.1 we repeat the use of the TOTS method for a finer and finer grid. The ripple region does appear to be shrinking in size. However, while ψ and $\frac{\partial\psi}{\partial R}$ seem to behave, $\frac{\partial^2\psi}{\partial R^2}$ is tending towards singular behavior. This creates a problem, for if ψ did exist at the light cylinder, $\frac{\partial^2\psi}{\partial R^2}$ (and higher derivatives) would have to exist there too (see Appendix D).

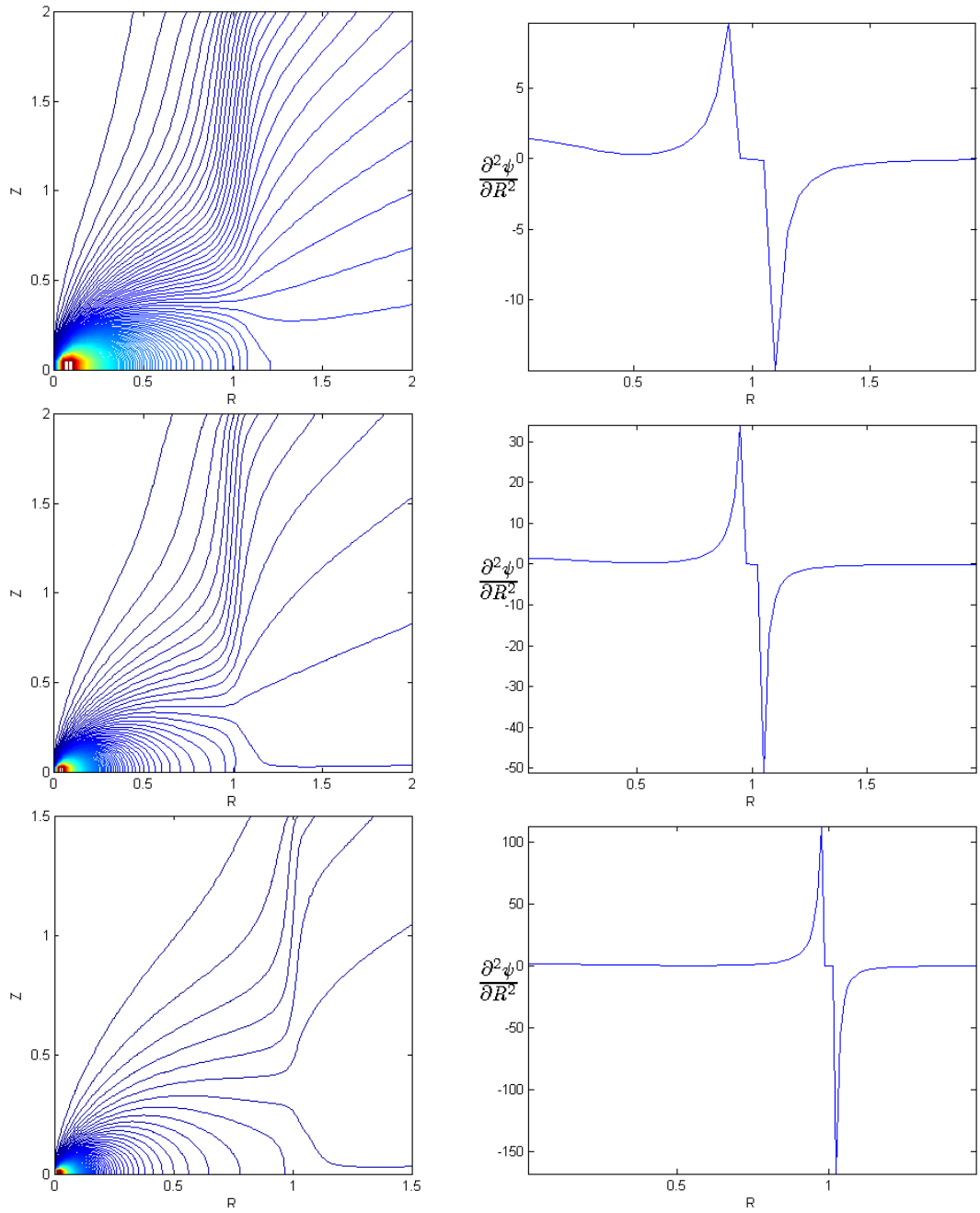


Figure 5.1: TOTS solutions with $\psi_{eq} = 1.0$ using the Altered TOTS method. The grid spacing in both directions is equal, and is 0.05, 0.025, and 0.0125. The second derivative vs. R is plotted next to the corresponding solution and is taken at $Z = 1$.

Do we believe the new method introduced in [3] was wrong? On the contrary, we believe it works exactly as advertised. Make a guess for $F(\psi)$, and then the method will solve for ψ , *whatever* the answer may be. There is no smoothing or iterative correcting here. You get whatever answer the equation has with that guess of $F(\psi)$, singular or otherwise.

One might wonder what exactly the takeaway message is. There are two ways to interpret this. The strict mathematical interpretation is that the guess of $F(\psi)$ made in [3] was wrong. Evidently, their guess corresponds to a singular solution, one that can exist on one side of the light cylinder or the other, but cannot go across and exist in all space. A solution cannot be “almost smooth.” CKF solution is smooth, and TOTS solution is not.¹

However, there is a more optimistic outlook. Their guess of $F(\psi)$ is meant to be an approximation of the (iteratively generated) CKF $F(\psi)$, and their solutions do look like CKF solutions. Furthermore, if given an exact form of $F(\psi)$ that corresponds to a known smooth solution, the method does return that smooth answer. So the method introduced in [3] clearly has some uses.

Now consider this observation. If you take a TOTS answer, and take the grid of ψ values and feed it as an initial guess to the Altered CKF method, then the answer is iteratively corrected to the CKF answer. Indeed, it is believed that this answer is always a unique answer. But this may hide valuable information. Using the TOTS method with an incorrect guess of $F(\psi)$ will give the actual, singular answer. The severity of the singular behavior could be an indication of how far off one is from a correct answer, and the lack of singular behavior

¹Note that CKF plots of $F(\psi)$ appear to be well-behaved everywhere, and [3] guessed $F(\psi)$ as a polynomial, which is ∞ -differentiable. By Appendix D, if they did find a smooth solution, all the derivatives of ψ with respect to R would have to exist at the light cylinder, ruling out any jumps in ψ or any of the R -derivatives.

could suggest that the guess is correct. In contrast, CKF seems to always give the same answer, no matter what you guess.²

With this in mind, in the following chapter, we show one way that the Altered TOTS method could be used as an investigative tool.

²Instead of initializing $F(\psi)$ to 0, one could initialize to a guess of $F(\psi)$, and then CKF would still reveal the number of iterations to converge, which may give some information. However, this would not be as straightforward to utilize.

CHAPTER 6

ADDITIONAL CRITIQUE ON JETS SOLUTIONS

Reconsider the guess of [4]:

$$H(\psi) = \frac{k_H}{2} \psi \left(\beta \frac{\psi}{\psi_{eq}} - 2 \right) \quad (6.1)$$

$$F(\psi) = \frac{k_H^2}{2} \psi \left(\beta \frac{\psi}{\psi_{eq}} - 1 \right) \left(\beta \frac{\psi}{\psi_{eq}} - 2 \right) \quad (6.2)$$

In this chapter we take this guess for all $0 \leq \psi < \psi_{eq}$. It was previously assumed that β was a constant. Here, we ask if considering β as a function of ψ (and k_H) will shed any light on this case. This idea was first conceived by noticing a curious observation. In some simulations using the Altered TOTS method, the solution would be singular at the light cylinder, but would have one very specific ψ contour that would pass through smoothly. Further, the value of this “lone contour” would change when β was changed. By finding pairs of ψ and β values, the hope was to derive a functional form for $\beta(\psi)$ that when used, would ensure all ψ contours would pass smoothly through the light cylinder.¹

Figure 6.1 and Table 6.1 show the points we found. Of course, there is an inherent error in using the above form of $F(\psi)$, since that formula assumes $\frac{d\beta}{d\psi} = 0$. One would have to vary both β and $\frac{d\beta}{d\psi}$ in the corrected $F(\psi)$ formula to remedy this. Nevertheless, the result of this “lone contour method” (see Appendix C) was both comforting and disappointing. We tried the functional fit:

$$\beta = \frac{2}{k_H} + 2 \left(\frac{\psi_{eq}}{\psi} \right) \left(1 - \frac{2}{k_H} \right) \quad (6.3)$$

¹There can be more than one “lone contour.” For example, if a parameter p really is $(\psi - 4)(\psi - 5)$, then if we set $p = 2$, both $\psi = 3$ and $\psi = 6$ contours will be smooth. We still stand by the name because these contours are usually separate from each other and are surrounded by singular contours, so they would both be alone.

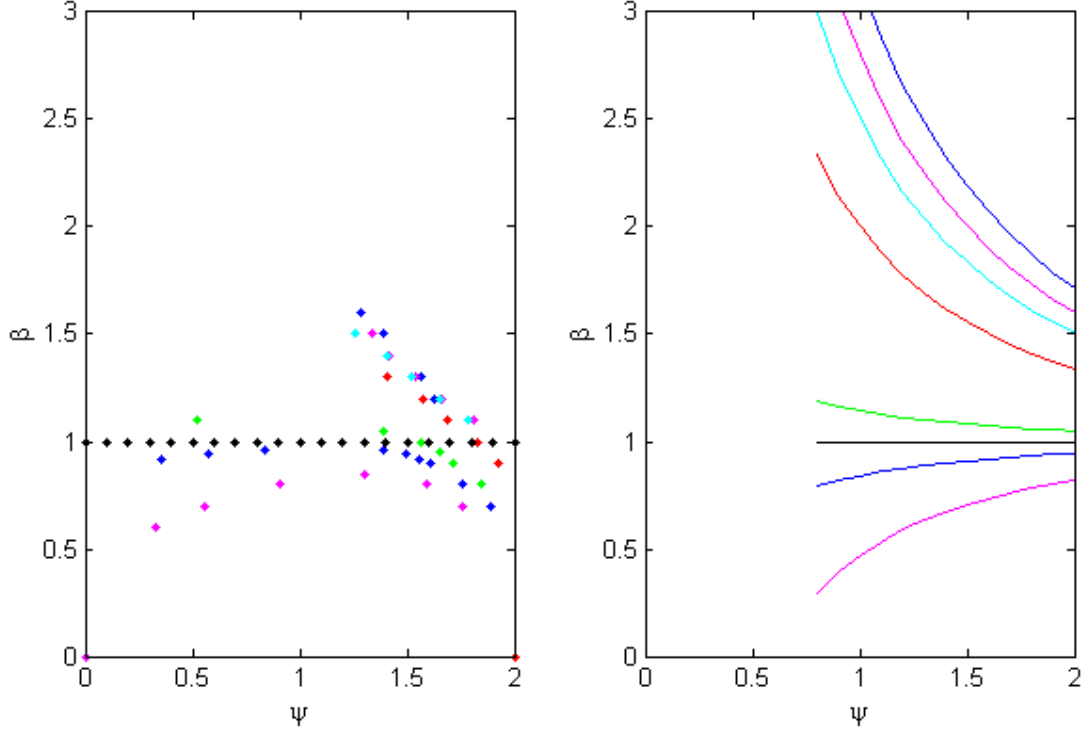


Figure 6.1: β vs. ψ , with $\psi_{eq} = 2$. From bottom to top, $k_H = 1.7$ (magenta), 1.9 (blue), 2.0 (black), 2.1 (green), 3.0 (red), 4.0 (cyan), 5.0 (magenta), and 7.0 (blue). $k_H = 2$ is the monopole solution ($\beta = 1$). The left plot is data estimated from the lone contour method, and the right plot is of Equation 6.3. See Table 6.1 for the complete list of data.

Which when plugged into $H(\psi)$ simply gives $H(\psi) = \psi(\frac{\psi}{\psi_{eq}} - 2)$, or the old monopole answer. This is identically true whether or not k_H is a function of ψ .

Now, we know this functional fit must work (the monopole answer is well established as being a valid solution), and it seems like the functional form of $\beta(\psi)$ that will work is unique. The obvious conclusion to jump to is that this fit that gives the monopole answer is the only fit that will work, ruling out any jet-like answers. If this is true, this would mean that if there is a new, non-monopole solution, it is not of the form that [4] guessed. As we allowed both k_H and β to

vary, this means an entire 2-dimensional space of parameters is knocked out.

The only problem is that the fit does not do a very good job of passing through the data points. $k_H \leq 2$ is plausible, but $k_H > 2$ values have a larger and larger gap from the fitted curve as k_H increases. Further, as ψ approaches ψ_{eq} , the data points seem to keep decreasing, perhaps to a vertical asymptote. But we can easily see that using our fit:

$$\beta(\psi_{eq}) = 2 \left(1 - \frac{1}{k_H} \right) \quad (6.4)$$

Which has very different behavior. Our interpretation of this is that the fit would work, except that one needs to vary both β and $\frac{d\beta}{d\psi}$ when searching for lone contours. After all, while a function and its derivative are obviously related, pointwise they are independent. Also, the bottom boundary artificially makes ψ_{eq} a smooth contour, which could skew the data points near ψ_{eq} .

There is one more parameter that we have already seen, the “ r ” in the TOTS $F(\psi)$. Revisiting Figures 4.4 and 4.5, we can see that despite the ripples at the light cylinder, there are contour lines that are able to pass through smoothly. These cases only vary ψ_{eq} , so we would expect the smooth $\frac{\psi}{\psi_{eq}}$ to be the same throughout all the simulations we have presented. We estimate some pairs of (ψ_{eq}, ψ) to be (0.5, 0.49299), (0.8, 0.79135), and (1.2, 1.19720), which all have a ratio close to 1. This was done with $r = 0.5$, so perhaps by varying r and finding this ratio, we could derive a relationship $r(\psi)$.

The primary reason for interest is that TOTS is approximating the CKF solution, which we *know* exists. So it really could be that finding $r(\psi)$ will help determine a closed form of the CKF solution. In the jets case, there was no definitive evidence that anything other than the monopole existed, so in retrospect it was not surprising that nothing new was found.

Table 6.1: Data obtained from the lone contour method. k_H and β are fixed in the program, while ψ is found from running the simulation.

k_H	ψ	β	k_H	ψ	β	k_H	ψ	β
7.0	1.2862	1.60	2.1	0.5194	1.10	1.0	1.5181	0.10
	1.3880	1.50		1.3889	1.05		1.2752	0.00
	1.4205	1.40		1.5625	1.00		1.7311	0.00
	1.5630	1.30		1.6537	0.95		1.1473	-0.10
	1.6239	1.20		1.7174	0.90		1.8280	-0.10
5.0	1.3358	1.50		1.8462	0.80		1.0404	-0.20
	1.4204	1.40	1.9	0.8405	0.96		1.9371	-0.20
	1.5413	1.30		1.3865	0.96		0.9499	-0.30
	1.6628	1.20		1.4937	0.94		0.8710	-0.40
	1.8135	1.10		0.5724	0.94		0.8102	-0.50
4.0	1.2560	1.50		0.3584	0.92		0.7476	-0.60
	1.4113	1.40		1.5578	0.92		0.6961	-0.70
	1.5214	1.30		1.6053	0.90		0.6474	-0.80
	1.6496	1.20		1.7615	0.80		0.6034	-0.90
	1.7839	1.10		1.8912	0.70		0.5619	-1.00
3.0	1.4046	1.30	1.7	1.2993	0.85	0.3	1.7811	-4.50
	1.5769	1.20		0.9098	0.80		1.3922	-4.50
	1.6879	1.10		1.5941	0.80		1.2471	-5.00
	1.8313	1.00		0.5570	0.70		1.0449	-6.00
	1.9221	0.90		1.7554	0.70		0.9152	-7.00
				0.3300	0.60		0.8054	-8.00
							0.7302	-9.00
							0.6405	-10.00
							0.5653	-11.00

CHAPTER 7

CONCLUSIONS

The altered methods augment the original methods by refining them and removing unnecessary steps which were introduced to accommodate numerical relaxation. We successfully have replicated prior results, and we strongly advocate that future researchers consider using Newton's method in place of relaxation, especially when using the TOTS method. We also see how the two ideas of CKF and TOTS have clearly different uses, and how the more recent TOTS method, rather than competing with CKF, can branch off into different investigative paths by using its unique ability to reveal, rather than correct and hide, singular solutions.

APPENDIX A

BOUNDARY CONDITIONS

We previously argued that in Equation 3.1, it is OK if coefficient a_5 vanishes. However, we must be careful about the boundary conditions. Although a small detail, it is important not to overlook a subtlety in dealing with boundary conditions where the term we wish to solve for does not exist because its coefficient vanishes. Using the example of the left boundary, consider these two cases:

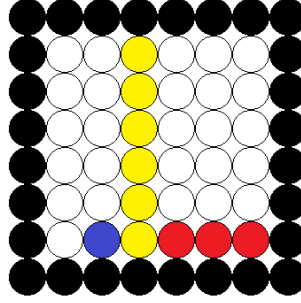
1. If the left boundary depends on the boundary point to the left of the edge point, then one can write:

$$b_1 U_{j-1,k} + b_2 U_{j+1,k} + b_3 U_{j,k-1} + b_4 U_{j,k+1} + b_5 U_{j,k} = b_6$$

Where b_1 will be nonzero to reflect the dependence on $U_{j-1,k}$. Then we can simply solve for $U_{j-1,k}$.

2. If the left boundary does not depend on the boundary point to the left of the edge point, then b_1 will be zero, and one cannot solve for $U_{j-1,k}$. However, in general $U_{j-1,k}$ will appear in the instance of Equation 3.1 that we wish to solve. For such boundary conditions, we can simply forget Equation 3.1 and write one custom equation. So, for example, if the left edge had the boundary condition $\frac{\partial \psi}{\partial Z}|_{(R_0, Z_0)} = 0$, then the equation would be $U_{j,k+1} - U_{j,k} = 0$. Here, the custom equation is in place of the usual technique of taking Equation 3.1 and plugging in the boundary equation. For the particular boundary conditions we demonstrated, this issue does not arise, but other boundary conditions that have been proposed may have to deal with this.

More complicated edge boundary conditions can also be accommodated by custom equations. For example, CKF's original mandate was that ψ_{eq} be found iteratively, rather than being a fixed parameter. Consider the following grid:



The light cylinder is marked in yellow, and the blue grid point is located at (j^*, k^*) . To enforce CKF's mandate, we use Equation 3.1 at each red point, and plug in the custom boundary equation $U_{j,k-1} = U_{j^*,k^*}$.

The last boundary issue to consider is what happens at a corner. A corner point touches two boundaries, and typically this just means creating two boundary equations to eliminate both dependencies. Sometimes, however, there are complications. For example, with our choice of boundary conditions, consider the upper right corner. Both the top and right boundaries give you the same equation, $-jU_{j-1,k} + jU_{j+1,k} - kU_{j,k-1} + kU_{j,k+1} = 0$. One strategy would be to write something of the form $f_1(U_{j+1,k}, U_{j,k+1}) = f_2(U_{j-1,k}, U_{j,k-1}, U_{j,k})$ in an effort to eliminate both $U_{j+1,k}$ and $U_{j,k+1}$ with one equation. In general, however, there is no guarantee this would work.

The way we get around this is to note that, when dealing with the outer boundaries, we can really use any two boundary equations, as long as they are compatible with the actual boundary condition. So, just for this corner point,

take these equations instead:

Top Boundary:

$$\frac{\partial \psi}{\partial Z}|_{(R_0, Z_0)} = 0 \longrightarrow (0, 0, -1, 1, 0, 0) \quad (\text{A.1})$$

Right Boundary:

$$\frac{\partial \psi}{\partial R}|_{(R_0, Z_0)} = 0 \longrightarrow (-1, 1, 0, 0, 0, 0) \quad (\text{A.2})$$

This gives us two equations that can eliminate both boundary points. Note that these equations being true forces the actual boundary equations to be true.

APPENDIX B

SMOOTHING OVER THE LIGHT CYLINDER

In forming the nonlinear equations, a priori we do not have to give the light cylinder points any special treatment. However, even if the system of equations with our choice of boundary conditions converges to something, there is no guarantee that this “something” reached is smooth, or even a real solution at all. This is why simulation methods incorporate some sort of smoothing requirement. The original CKF method, for example, achieves a smooth solution through its iterative process. For us, we have overwritten the equations for points near the light cylinder.

One might wonder what would happen without any smoothing. Can the nonlinear equations be solved as is? Reconsider the overwrite that accommodated the light cylinder:

$$\text{If } N_{LC} - Q \leq j \leq N_{LC} + Q$$

$$\text{Then } (a_1, a_2, a_3, a_4, a_5, a_6) = (-1/2, -1/2, 0, 0, 1, 0)$$

Where $Q \leq -1$ would eliminate the overwrite completely (the condition would never be true and this would be skipped), $Q = 0$ would only affect the light cylinder itself, and $Q \geq 1$ forces smoothness further away ($Q > 1$ is usually not needed).

In every simulation up to this point, $Q = 1$ was used. But perhaps this smoothing step is completely unnecessary. Eliminating the smoothness overwrite completely would clean up the new methods further. In Figures B.1 and B.2 we present various solutions, with $Q = -1, 0$, and 1 . All simulations in this section are done with a grid spacing of 0.025 in both directions, and use an

80x80 grid.¹

The Altered CKF method seems to have slight trouble at $Q = -1$. Although some answers generally appear to be smooth over the light cylinder, it seems we do need additional smoothing to get rid of some ripples. However, the need for additional smoothing is an illusion. CKF demands that ψ at the light cylinder is the average of the values to the immediate left and right of it, and this is precisely the only difference between $Q = -1$ and $Q = 0$. So the Altered CKF method does not really need any additional smoothing. Using $Q = -1$ would just create a conflict of goals, tugging the solution in two different directions needlessly. We simply use $Q = 0$ as an easy way to enforce a part of the method without causing unnecessary complication.

As for the Altered TOTS method, we have previously argued that the TOTS solutions were indeed singular at the light cylinder, and for all values of Q we can see trouble. Furthermore, using an estimate of $F(\psi)$, as in the Null Sheet case, will also produce ripples for all Q . This makes sense, as the estimate itself is not likely to be an exact allowed function, so there will not be an exact smooth solution to go with it. We can also see how using a Q value that is too high may smooth over problems in the answer. But if the Altered TOTS method is used in a case where there is known to be a solution, then the method will work with $Q = -1$. For example, the monopole picture looks smooth. Also, if one uses the Altered CKF method to get an answer (with any Q), but then uses the iteratively found $F(\psi)$ in the Altered TOTS method with $Q = -1$, then you will get the exact same answer despite removing the smoothing. In other words, absolutely

¹One might ask from a practical point of view why this matters at all. If you already know you will get a smooth answer, then using some smoothing is a sensible idea. However, if a solution is singular, smoothing can hide that fact. The point here is to show that there exists the option not to use smoothing.

no smoothing is required. Altered TOTS can simply solve the equations, exactly as they are, to get an answer. Smoothing is simply a convenience to get the answer in less iterations.

So in essence, both altered methods do not require much special treatment of the light cylinder (above and beyond what the CKF method already demanded). Any smoothing introduced is only to speed up simulations. It is comforting to know there is no theoretical need for it. The only real requirement we have not addressed is that the light cylinder lands exactly on the grid.

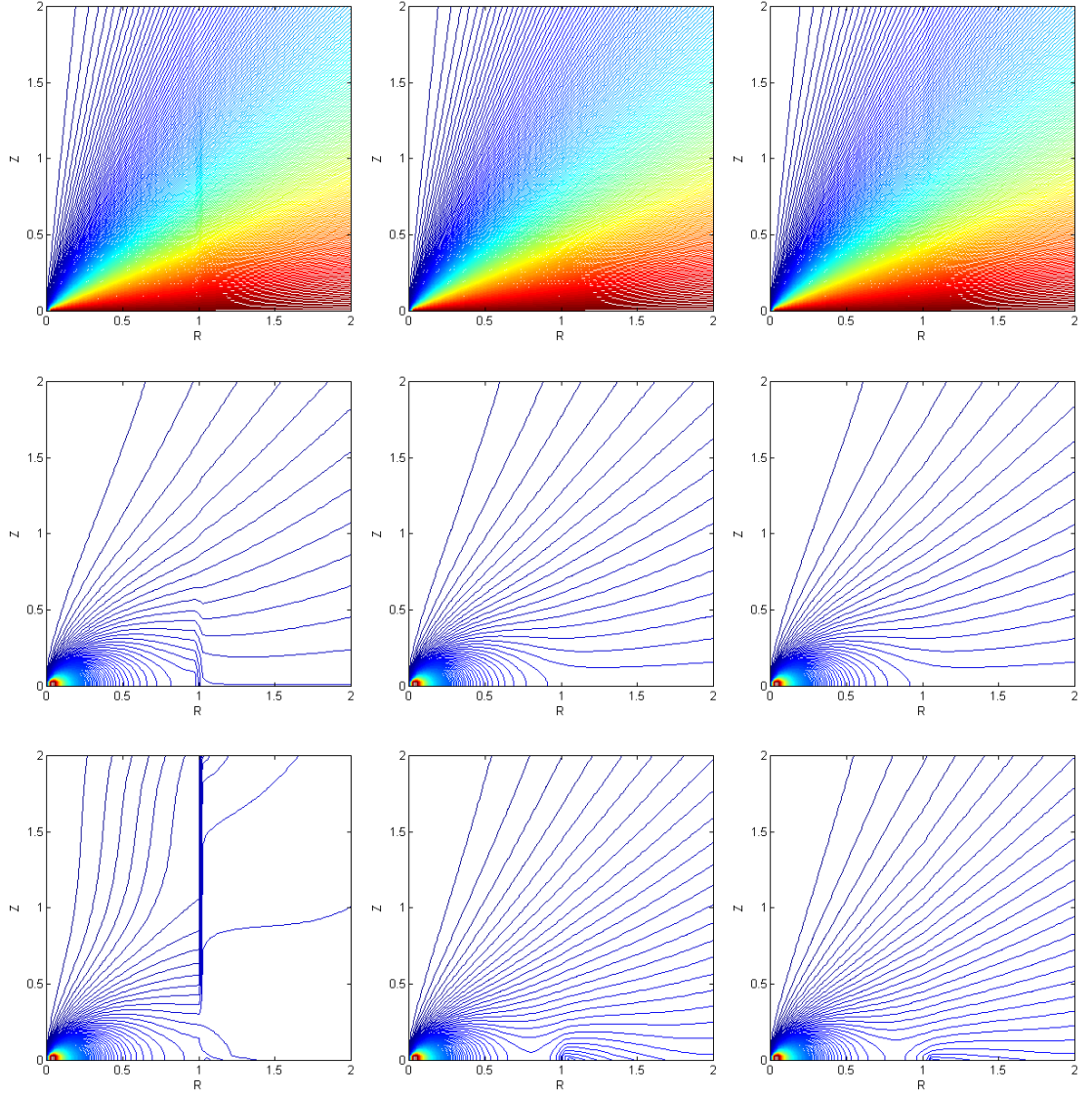


Figure B.1: Various solutions with the Altered CKF method. Top row is monopole ($\psi_{eq} = 1$), then CKF ($\psi_{eq} = 1.28$), then Null Sheet ($\psi_{eq} = 2$). Left column is $Q = -1$, then $Q = 0$, then $Q = 1$.

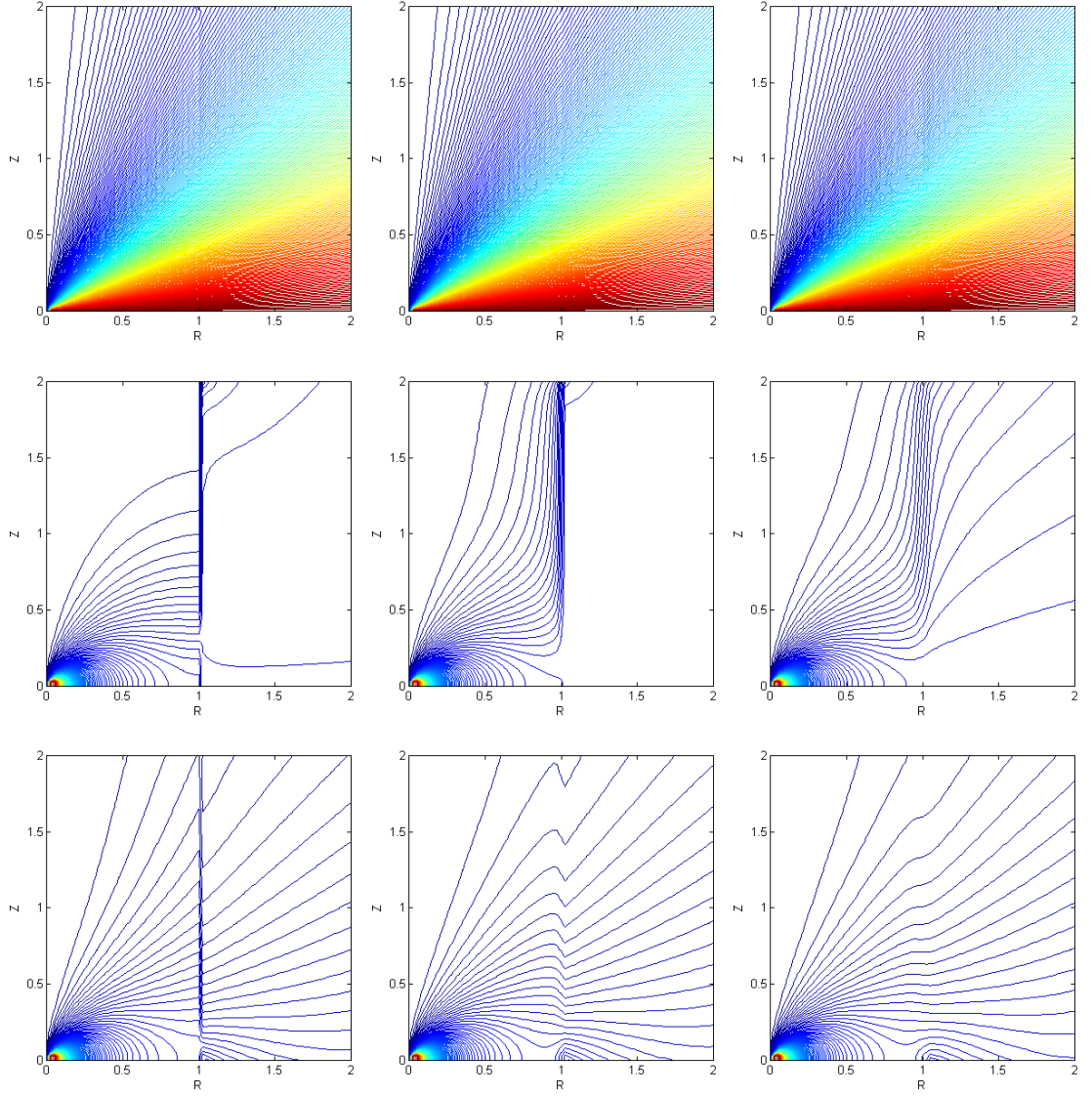


Figure B.2: Various solutions with the Altered TOTS method. Top row is monopole ($\psi_{eq} = 1$), then TOTS ($\psi_{eq} = 1.28$), then Null Sheet ($\psi_{eq} = 2$). Left column is $Q = -1$, then $Q = 0$, then $Q = 1$.

APPENDIX C

FUTURE OPPORTUNITIES

C.1 The Lone Contour Method

A new investigative method:

Write any guess for $F(\psi)$ that gives a singular answer and that has at least one parameter that can be varied. Run simulations with the Altered TOTS method and search for “lone” smooth contours by varying the parameters. Use this data to derive functional relationships between the parameters and smooth ψ values, in an effort to correct the original $F(\psi)$ guess.

Of course, one must exercise caution. For example, when we applied this method to the jets case, β was a function of both k_H and ψ . One could imagine that having many parameters could make this complicated. Another consideration is that it is not easy to estimate the lone contour’s value. We show how we identify it in Figure C.1, but a lot of estimation is required. Our grid spacing of 0.0125 is not small enough, but we do feel that a finer grid would eventually provide any level of precision desired. A further reaching grid would also be useful to avoid undue influence from the boundary.

One thing this method has going for it is that we can take advantage of our shortcut of keeping J^{-1} fixed. Since we are only changing $F(\psi)$ and not the grid itself, we can simply read J^{-1} from a file, which is much faster than calculating a matrix inverse each simulation. Further, for a last bit of speed, we can run multiple simulations with J^{-1} in memory, and use the end result of one simulation as the initial guess for the next.

One last reminder is that one need not choose a form of $F(\psi)$ that generalizes the monopole. The jets guess does generalize it (reducing to the monopole for $k_H = 2$), but the TOTS guess does not. Whether or not there are separate families of solutions, or one correct general functional form is an open question.

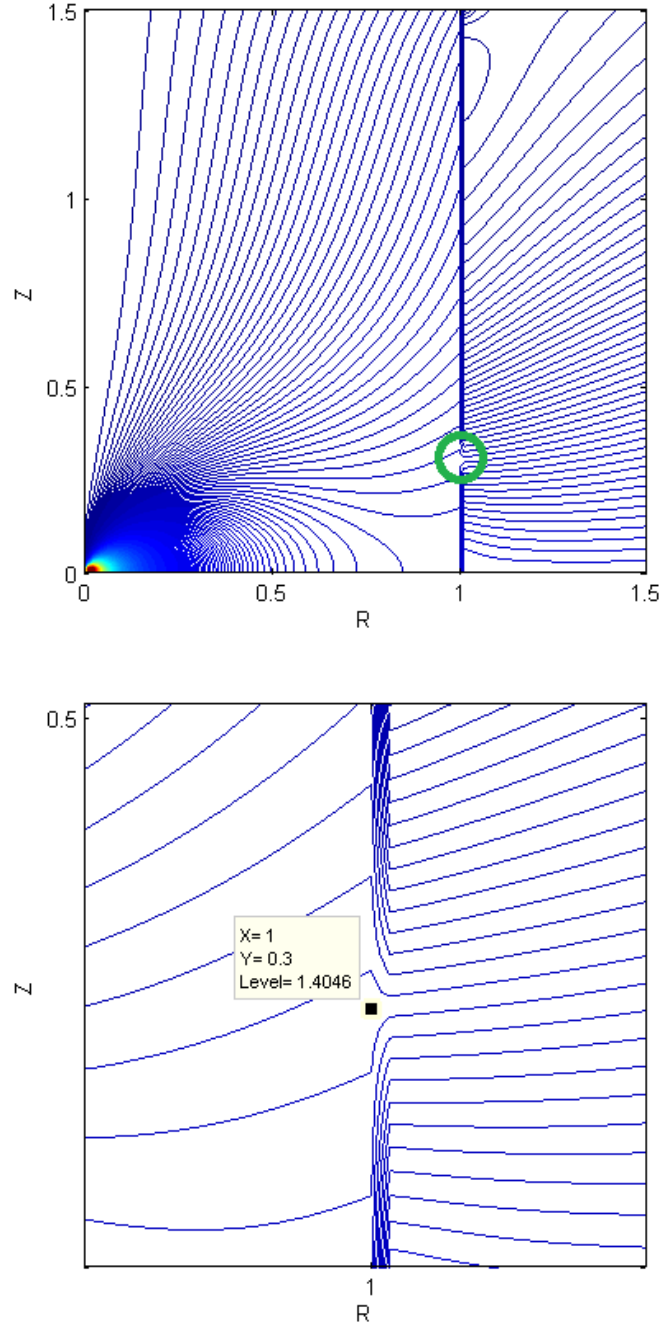


Figure C.1: Jets simulation ($Q = -1$, $\psi_{eq} = 2$, $k_H = 3$, $\beta = 1.3$), demonstrating a lone contour. The entire light cylinder is singular except for one location. Our grid spacing is 0.0125 in both directions, but a higher resolution is recommended for further study.

C.2 Unique Solution?

If we were to fix a boundary, choose a method (CKF or TOTS, original or altered), and reach a solution, is that solution unique? So far, the evidence says yes. However, most of that evidence is the observation that numerical relaxation reaches the same answer despite different initial values. However, there is an opportunity to get stronger evidence.

Newton's method reveals a new way of looking at this problem, i.e. as a set of nonlinear equations. Although Newton's method has converged to a unique solution for every problem considered thus far, writing the nonlinear equations provides an invitation for other solving techniques that could search for additional physical solutions. One possible avenue that was considered was homotopy, a process which utilizes Newton's method to trace out paths to each solution to the nonlinear equations, rather than simply converging to the one closest to the initial values. One could discard solutions with any nonpositive ψ value, and then look at any solutions that remain. This technique could either find other physical solutions, or simply provide further evidence of their nonexistence.

APPENDIX D

MATHEMATICAL PROOF

Let $C(n) \equiv$ "Let $(1, Z_0)$ be a point on the light cylinder ($Z_0 > 0$) where ψ exists. If the n^{th} derivative of $F(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$, then for $m \in \mathbb{Z}$, $\frac{\partial^m \psi}{\partial R^m}$ exists at $(1, Z_0)$ if $1 \leq m \leq n + 1$."

Claim: For $n \in \mathbb{Z}_{\geq 0}$, $C(n)$ is true.

Proof: We use induction on n :

Base case: Prove $C(0)$

$(1, Z_0)$ is on the light cylinder, so the pulsar equation will give us the usual smoothness condition there:

$$\frac{\partial \psi}{\partial R} = \frac{1}{2} F(\psi) \tag{D.1}$$

Now, ψ exists at $(1, Z_0)$, and $F(\psi)$ exists at $\psi(1, Z_0)$, so simply plug in $\psi(1, Z_0)$ into $F(\psi)$ to get $F(\psi)$ at $(1, Z_0)$. Since this must exist, the other side of the smoothness equation must also exist. Hence, $\frac{\partial \psi}{\partial R}$ exists at $(1, Z_0)$.

Inductive Step: For $n > 0$, prove $C(n - 1) \longrightarrow C(n)$

Here, we are assuming the n^{th} derivative of $F(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$, and this also means that the $(n - 1)^{th}$ derivative of $F(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$. By the inductive assumption, we get to conclude that $\frac{\partial^m \psi}{\partial R^m}$ exists at $(1, Z_0)$ for $1 \leq m \leq n$. We just need to show that $\frac{\partial^{n+1} \psi}{\partial R^{n+1}}$ also exists at $(1, Z_0)$.

Once again consider the smoothness condition. With additional differentiation:

$$\frac{\partial^{n+1}\psi}{\partial R^{n+1}} = \frac{1}{2} \frac{\partial^n F(\psi)}{\partial R^n} \quad (\text{D.2})$$

To apply the chain rule a variable number of times, we take advantage of a form of Faà di Bruno's formula:

$$\frac{\partial^n F(\psi)}{\partial R^n} = \sum_{j=1}^n \frac{\partial^j F(\psi)}{\partial \psi^j} B_{n,j} \left(\frac{\partial \psi}{\partial R}, \dots, \frac{\partial^{n-j+1} \psi}{\partial R^{n-j+1}} \right) \quad (\text{D.3})$$

Where $B_{n,j}$ are the Bell polynomials of combinatorial mathematics. The only detail about them that matters is that if the arguments all exist, then this polynomial will also exist.

Since $1 \leq j \leq n$, by our assumption and by plugging in $\psi(1, Z_0)$, all of the required derivatives of $F(\psi)$ will exist at $(1, Z_0)$. Also, for $1 \leq j \leq n$, we have $1 \leq n - j + 1 \leq n$ and as we previously argued, all of the required derivatives of ψ will exist at $(1, Z_0)$. So all terms on the right hand side exists at $(1, Z_0)$, meaning $\frac{\partial^n F(\psi)}{\partial R^n}$ exists at $(1, Z_0)$, and thus $\frac{\partial^{n+1} \psi}{\partial R^{n+1}}$ exists at $(1, Z_0)$, completing the proof.

Notes about the proof:

- 1: We previously fixed the light cylinder to be $R = 1$.
- 2: For convenience in writing the proof, the “zeroth derivative” of a function is understood to just be the function itself.
- 3: The claim is written in terms of $F(\psi)$. However it is straightforward to show that for $n \geq 0$:
 “The n^{th} derivative of $F(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$.” \longleftrightarrow
 “The $(n + 1)^{\text{th}}$ derivative of $H(\psi)$ with respect to ψ exists at $\psi(1, Z_0)$.”

APPENDIX E

$H(\psi)$ DICTIONARY

For reference, and especially for those who would like to employ the lone contour method, we would like to offer a centralized reference of $H(\psi)$ guesses (mostly from other authors), and any solutions known. This will give perspective on how these guesses are related, and which parameters they all have and where there is room to generalize more. Note that not all of these guesses have physical boundary conditions on $H(\psi)$ applied, but we do assume that ψ is always greater than or equal to the constant term. Also, we remind the reader that if $\psi(R, Z)$ and $F(\psi)$ fit the pulsar equation, then $k_1\psi(R, Z + k_2) + k_3$ and $k_1F(\psi - k_3)$ must also fit.¹

We have attempted to list the first source that proposed, in some way, each solution in our dictionary. Solutions marked “Original,” to the best of our knowledge, are not mentioned in prior literature. The trivial solution is obvious so we will refrain from giving credit. Also, since the CKF solution is only known implicitly, it does not yet have a place in our dictionary.

¹We do not show shifts of the star origin in the formulas explicitly, but from the pulsar equation, and as pointed out by [8], solutions should be unaffected by a shift in the Z coordinate. In other words, if we solve the pulsar equation, we are really finding a solution representing a star centered at any Z location, but at R=0. One could easily alter the pulsar equation by shifting the R coordinate. Then we could get answers centered anywhere.

“Trivial”:

$$\psi = c_1 Z + c_2$$

$$H(\psi) = [\text{Any Constant}]$$

$$F(\psi) = 0$$

“Real Monopole” [5]:

$$\psi = c_1 + \frac{c_2 Z}{\sqrt{R^2 + Z^2}} \quad (c_2 \neq 0)$$

$$H(\psi) = \pm c_2 \left(\frac{\psi - c_1}{c_2} + 1 \right) \left(\frac{\psi - c_1}{c_2} - 1 \right)$$

$$F(\psi) = 2(\psi - c_1) \left(\frac{\psi - c_1}{c_2} + 1 \right) \left(\frac{\psi - c_1}{c_2} - 1 \right)$$

“TOTS” [3]:

$$\psi = \psi_{eq}(c_1 + [\text{Unknown Non-constant Terms}]) \quad (r \neq 0)$$

$$H(\psi) = \pm \left(\frac{\psi - c_1}{\sqrt{2r}} \right) \sqrt{\left(\frac{\psi - c_1}{\psi_{eq}} \right)^2 - \frac{4}{3}(r+1) \left(\frac{\psi - c_1}{\psi_{eq}} \right) + 2r}$$

$$F(\psi) = \left(\frac{\psi - c_1}{r} \right) \left(\frac{\psi - c_1}{\psi_{eq}} - 1 \right) \left(\frac{\psi - c_1}{\psi_{eq}} - r \right)$$

“Jets” [4]:

$$\psi = \psi_{eq}(c_1 + [\text{Unknown Non-constant Terms}]) \quad (k_H \neq 0)$$

$$H(\psi) = \pm \frac{k_H}{2}(\psi - c_1) \left(\beta \frac{\psi - c_1}{\psi_{eq}} - 2 \right)$$

$$F(\psi) = \frac{k_H^2}{2}(\psi - c_1) \left(\beta \frac{\psi - c_1}{\psi_{eq}} - 1 + \frac{\beta'}{2} \frac{(\psi - c_1)^2}{\psi_{eq}} \right) \left(\beta \frac{\psi - c_1}{\psi_{eq}} - 2 \right)$$

“Null Sheet” [7]:

$$\begin{aligned}\psi &= \psi_{eq}(c_1 + [\text{Unknown Non-constant Terms}]) \\ H(\psi) &= \pm 1.07(\psi - c_1) \left(2 - \frac{\psi - c_1}{\psi_{eq}}\right) \left(1 - \frac{\psi - c_1}{\psi_{eq}}\right)^{0.4} \\ F(\psi) &= \frac{2}{5}(1.07)^2(\psi - c_1) \left[6 \left(\frac{\psi - c_1}{\psi_{eq}}\right)^2 - 12 \frac{\psi - c_1}{\psi_{eq}} + 5\right] \\ &\quad \left(2 - \frac{\psi - c_1}{\psi_{eq}}\right) \left(1 - \frac{\psi - c_1}{\psi_{eq}}\right)^{-0.2}\end{aligned}$$

“Imaginary Monopole” (Original):

$$\begin{aligned}\psi &= c_1 + \frac{c_2}{\sqrt{R^2 + Z^2}} \quad (c_2 \neq 0) \\ H(\psi) &= \pm(\sqrt{-1})(\psi - c_1) \left(\frac{\psi - c_1}{c_2}\right) \\ F(\psi) &= -2(\psi - c_1) \left(\frac{\psi - c_1}{c_2}\right)^2\end{aligned}$$

“No Z dependance” (Original):

$$\begin{aligned}\psi &= c_1 R^j + c_2 \quad (j \neq 0 \ \& \ c_1 \neq 0) \\ H(\psi) &= \begin{cases} \pm \sqrt{(\psi - c_2)^2 + 2c_1^2 \text{Log}_e(c_2 - \psi)} & j = 1 \\ \pm j(\psi - c_2) \sqrt{1 - \left(\frac{j-2}{j-1}\right) \left(\frac{\psi - c_2}{c_1}\right)^{-\frac{2}{j}}} & \text{else} \end{cases} \\ F(\psi) &= (\psi - c_2) \left(j^2 - j(j-2) \left(\frac{\psi - c_2}{c_1}\right)^{-\frac{2}{j}}\right)\end{aligned}$$

“Simple R and Z dependance” [8]:

$$\begin{aligned}\psi &= c_1 R^2 Z + c_2 \quad (c_1 \neq 0) \\ H(\psi) &= \pm 2(\psi - c_2) \\ F(\psi) &= 4(\psi - c_2)\end{aligned}$$

APPENDIX F

SOURCE CODE

We provide a complete listing of the code used. Eight files represent the main structural code, while eighteen files each represent a distinct simulation case. All code is in the C language, and was compiled with Pelles C 7.00.347.

F.1 Main Code

This section includes the source code for the main structure of the simulation.

- **matrix.c** is the main code and entry point. This contains a control panel which allows the user to change the grid parameters and other settings in one centralized location. The main program loop is here, along with all the data writing to files at the end of the simulation.
- **matrix_build.c** fills in the entries to the matrix and vector that represent the linear parts of the pulsar equation. This code assumes that variable relationships are only with nearest neighbors.
- **inverse.c** calculates the inverse of a matrix. Our shortcut ensures that we only need to call this once.
- **equation_solver.c** and **equation_solver_2case.c** use Newton's method to solve the nonlinear pulsar equation. Any nonlinearity comes from $F(\psi)$. The latter file allows the accommodation of double simulations, which start with one case and then switch to another, such as using the CKF method and then feeding the $F(\psi)$ values generated into the TOTS method.
- **reset_ckf.c** and **reset_tak.c** reset part of the bottom boundary to accommodate Null Sheet cases.
- **read_matrix.c** reads the inverse Jacobian matrix from file so that it does not have to be calculated, saving a significant amount of time. The simulation parameters used when the matrix was written cannot be changed, or else reading the matrix for that new simulation will result in nonsensical results. Also, this is not recommended for any Null Sheet cases.

F.1.1 matrix.c

```
1 //Matrix Builder
2 //Applies Newton's method to differential equations using a matrix.
3 //Corresponds to Vidal and Lovelace 2014 (Draft) and "New Treatment
  of the Pulsar Equation" (Master's Thesis).
4 //Created September 25, 2013 By Michael Joseph Vidal.
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <conio.h>
9 #include <io.h>
10 #include <math.h>
11 #include <string.h>
12 #include <time.h>
13
14 //Definitions for the types of simulations found in the program.
15 #define MONOPOLE 1
16 #define STANDARD 2
17 #define JETS 3
18 #define NULLSHEET 4
19
20 //This region of code serves as a control panel for the rest of the
  program. Comment or uncomment macros to get desired behavior.
21 typedef double T; //Choose your precision (float or double)
22
23 /*
24 Makes program read the matrix "J_INVERSE" from file instead of
  calculating it.
25 EVERY grid setting below must EXACTLY match the ones used in the file
  read.
26 READ may not work for Null Sheet cases.
27 */
28 // #define READ
29
30 #define SHORTCUT //Assumes constant Jacobian Matrix.
31 T V_FRAC=0.1; //Similar to relaxation parameter.
32
33 //These only affect the jets case. So far, only ckf_jets.c has this
  implemented.
34 #define JETS1 //Use boundary used by Takamori in jets case (this one
  is the typical choice).
35 // #define JETS2 //Use boundary used by Lovelace in jets case.
36
37 #define SMOOTH 1 //Amount of smoothing w.r.t. the light cylinder.
  SMOOTH 1 is the typical value.
38
39 #define NUM_ITERATIONS_MAX 100000 //Maximum number of iterations,
  regardless of convergence.
40
41 //Grid Parameters.
42 #define N_MAX_X 42
```

```

43 #define N_MAX_Y 42
44 #define N_LC 20
45 #define DX 0.05 //Make sure N_LC*DX=1.
46 #define DY 0.05
47 #define N_S_X 2 //Make sure the star is square i.e. DX*N_S_X=DY*N_S_Y
48 #define N_S_Y 2
49
50 //Other simulation parameters.
51 #define RATIO 0.5 //Only used in TOTS case.
52 #define P_OP 1.0 //PSI EQUATORIAL.
53
54 int toggle=0; //Only used in double simulation cases.
55 T BETA=1.0; //Only used in jets case. (Defining it here allows us to
    change it mid simulation.
56
57 //Sizes of various objects used in the program.
58 const int SIZE_T=sizeof(T);
59 const int SIZE_T_X=(N_MAX_X-2)*sizeof(T);
60 const int SIZE_T_Y=(N_MAX_Y-2)*sizeof(T);
61 const int SIZE_T_X_Y=(N_MAX_X-2)*(N_MAX_Y-2)*sizeof(T);
62 const int SIZE_T_XY=(N_MAX_X-2)*(N_MAX_Y-2)*sizeof(T);
63 const int SIZE_T_XY_XY=(N_MAX_X-2)*(N_MAX_X-2)*(N_MAX_Y-2)*(N_MAX_Y
    -2)*sizeof(T);
64
65 //Files that are written to. Some are extra files that may be useful
    for debugging or for secondary data.
66 FILE *fp_r,*fp_z,*fp_a,*fp_b,*fp_j,*fp_j2,*fp_hhp,*fp_hhpPRIME,*fp_v
    ,*fp_vxy,*fp_P_HHP,*fp_change;
67
68 T L_func[(N_MAX_X-2)];
69 T L_X[(N_MAX_X-2)*(N_MAX_Y-2)];
70 T L_Y[(N_MAX_X-2)*(N_MAX_Y-2)];
71 T CHANGE[(N_MAX_X-2)][(N_MAX_Y-2)];
72
73 T **A;
74 T **B;
75 T **J;
76 T **J_INVERSE;
77 T **HHP;
78 T **HHP_PRIME;
79 T *v0;
80 T *v1;
81 T *eq;
82 T **h;
83 T *p;
84
85 T **a1,**a2,**a3,**a4,**a5,**a6;
86 T *bL1,*bL2,*bL3,*bL4,*bL5,*bL6;
87 T *bR1,*bR2,*bR3,*bR4,*bR5,*bR6;
88 T *bB1,*bB2,*bB3,*bB4,*bB5,*bB6;
89 T *bT1,*bT2,*bT3,*bT4,*bT5,*bT6;

```

```

90 T ***c;
91
92 void initialize(void);
93 void star(void);
94 __forceinline void create_matrix(void);
95 void writeToFiles(void);
96 __forceinline int RN(int,int);
97 __forceinline int CN_X(int);
98 __forceinline int CN_Y(int);
99 __forceinline T f(int,int);
100 __forceinline void hhpSet(T**,T**);
101 void equationBuilder(int);
102 void inverse(T**,int,T**);
103 T** Make2DArray(int,int);
104 void resetCKF(void);
105 void resetTak(void);
106 void read(void);
107
108 /*
109 Case file choices. Include EXACTLY ONE of these to choose the case.
110 */
111 //Single simulation cases.
112 #include "ckf_monopole.c" //Pure Monopole case with CKF method.
113 //#include "ckf.c" //Pure CKF method from Contopoulos et al. 1999.
114 //#include "ckf_jets.c" //CKF case mixed with Jets case.
115 //#include "ckf_null.c" //Null Sheet case with CKF method.
116 //#include "tak_monopole_test.c" //Test of the Monopole case using
    the known answer.
117 //#include "tak_monopole.c" //Pure Monopole case with the TOTS method
    .
118 //#include "tak_monopole_jets.c" //Monopole case mixed with Jets case
    .
119 //#include "tak.c" //Pure TOTS method from Takamori et al. 2012.
120 //#include "tak_jets.c" //TOTS case mixed with Jets case.
121 //#include "tak_theory_jets.c" //Theoretical jets case.
122 //#include "tak_null.c" //Null Sheet case with TOTS method.
123
124 /*
125 Double simulation cases. There are two general situations:
126 You start with TOTS method, and then go to CKF method,
127 or, You start with CKF method, and then go to TOTS method.
128 Note: Doing TOTS first will give CKF the grid of PSI values. Doing
    CKF first will give TOTS the grid of HHP values.
129 */
130 //#include "ckf_monopole_jets.c" //CKF Monopole, then Jets.
131 //#include "ckf_tak.c" //CKF, then TOTS.
132 //#include "ckf_null_tak.c" //CKF Null Sheet, then TOTS.
133 //#include "tak_ckf.c" //TOTS, then CKF.
134 //#include "tak_ckf_jets.c" //TOTS, then CKF Jets.
135 //#include "tak_ckf_null.c" //TOTS, then CKF Null Sheet.
136 //#include "tak_theory_jets_ckf.c" //TOTS Theoretical Jets case, then
    CKF.

```

```

137
138 //The code that creates the matrix "A" and vector "b" for the linear
    system
139 #include "matrix_build.c"
140
141 //Additional code to reset part of the bottom boundary every
    iteration, which is neccessary only for Null Sheet cases.
142 #include "reset_ckf.c" //Only needed if ckf_null is an included case
143 #include "reset_tak.c" //Only needed if actual_null is an included
    case
144
145 //Code that calculates the inverse of a matrix.
146 #include "inverse.c"
147
148 //Code to solve the nonlinear equations.
149 #include "equation_solver.c" //Use with a "single simulation" case.
150 //#include "equation_solver_2case.c" //Use with a "double simulation"
    case.
151
152 //Code that reads the matrix "J_INVERSE" from file instead of
    calculating it.
153 #include "read_matrix.c"
154
155 int main(int argc, char *argv[])
156 {
157     //Memory allocation and initialization.
158     A=Make2DTArray( (N_MAX_X-2) * (N_MAX_Y-2), (N_MAX_X-2) * (N_MAX_Y
        -2));
159     B=malloc(SIZE_T_XY);
160     J=Make2DTArray( (N_MAX_X-2) * (N_MAX_Y-2), (N_MAX_X-2) * (N_MAX_Y
        -2));
161     J_INVERSE=Make2DTArray( (N_MAX_X-2) * (N_MAX_Y-2), (N_MAX_X-2) * (
        N_MAX_Y-2));
162     v0=malloc(SIZE_T_XY);
163     v1=malloc(SIZE_T_XY);
164     eq=malloc(SIZE_T_XY);
165     h=Make2DTArray( (N_MAX_X-2) * (N_MAX_Y-2), (N_MAX_X-2) * (N_MAX_Y
        -2));
166     p=malloc(SIZE_T_XY);
167
168     HHP=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
169     HHP_PRIME=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
170
171     a1=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
172     a2=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
173     a3=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
174     a4=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
175     a5=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
176     a6=Make2DTArray(N_MAX_X-2,N_MAX_Y-2);
177
178     bL1=malloc(SIZE_T_Y);
179     bL2=malloc(SIZE_T_Y);

```



```

180     bL3=malloc (SIZE_T_Y);
181     bL4=malloc (SIZE_T_Y);
182     bL5=malloc (SIZE_T_Y);
183     bL6=malloc (SIZE_T_Y);
184
185     bR1=malloc (SIZE_T_Y);
186     bR2=malloc (SIZE_T_Y);
187     bR3=malloc (SIZE_T_Y);
188     bR4=malloc (SIZE_T_Y);
189     bR5=malloc (SIZE_T_Y);
190     bR6=malloc (SIZE_T_Y);
191
192     bB1=malloc (SIZE_T_X);
193     bB2=malloc (SIZE_T_X);
194     bB3=malloc (SIZE_T_X);
195     bB4=malloc (SIZE_T_X);
196     bB5=malloc (SIZE_T_X);
197     bB6=malloc (SIZE_T_X);
198
199     bT1=malloc (SIZE_T_X);
200     bT2=malloc (SIZE_T_X);
201     bT3=malloc (SIZE_T_X);
202     bT4=malloc (SIZE_T_X);
203     bT5=malloc (SIZE_T_X);
204     bT6=malloc (SIZE_T_X);
205
206     c=malloc (4*SIZE_T_X_Y);
207     c[0]=Make2DArray (N_MAX_X-2,N_MAX_Y-2);
208     c[1]=Make2DArray (N_MAX_X-2,N_MAX_Y-2);
209     c[2]=Make2DArray (N_MAX_X-2,N_MAX_Y-2);
210     c[3]=Make2DArray (N_MAX_X-2,N_MAX_Y-2);
211
212     //Gives a rough idea of how much memory is used while the
        program is running.
213     printf ("SIZE T MBytes=%f\n", (T) SIZE_T/(1024*1024));
214     //Big stuff
215     printf ("Big Stuff\n");
216     printf ("MBytes A=%f\n", (T) SIZE_T_XY_XY/(1024*1024));
217     printf ("MBytes J=%f\n", (T) SIZE_T_XY_XY/(1024*1024));
218     printf ("MBytes J-1=%f\n", (T) SIZE_T_XY_XY/(1024*1024));
219     printf ("MBytes h=%f\n", (T) SIZE_T_XY_XY/(1024*1024));
220
221     //Small stuff
222     printf ("\nSmall Stuff\n");
223     printf ("MBytes B=%f\n", (T) SIZE_T_XY/(1024*1024));
224     printf ("MBytes v,eqBlock=%f\n", 3.0*SIZE_T_XY/(1024*1024));
225     printf ("MBytes p=%f\n", (T) SIZE_T_XY/(1024*1024));
226     printf ("MBytes hhpBlock=%f\n", 2.0*SIZE_T_X_Y/(1024*1024));
227     printf ("MBytes aBlock=%f\n", 6.0*SIZE_T_X_Y/(1024*1024));
228     printf ("MBytes bBlock=%f\n", 12.0*(SIZE_T_X+SIZE_T_Y)
        / (1024*1024));
229     printf ("MBytes cBlock=%f\n", 4.0*SIZE_T_X_Y/(1024*1024));

```

```

230
231 #ifndef READ
232     //Initialize everything to zero
233     for (int j=0; j<=(N_MAX_X-2)*(N_MAX_Y-2)-1; j++)
234     {
235         B[j]=0;
236         for (int k=0; k<=(N_MAX_X-2)*(N_MAX_Y-2)-1; k++)
237         {
238             A[j][k]=0;
239         }
240     }
241 #endif
242
243     for(int x=0; x<N_MAX_X-2; x++)
244     {
245         for(int y=0; y<N_MAX_Y-2; y++)
246         {
247             HHP[x][y]=0;
248             HHP_PRIME[x][y]=0;
249         }
250     }
251
252     //Prepare files. Note that some files are for secondary
253     information that is largely unused.
254     fp_r=fopen("r.dat", "w+"); //R Values.
255     fp_z=fopen("z.dat", "w+"); //Z Values.
256     fp_a=fopen("a.dat", "w+"); //A Matrix.
257     fp_b=fopen("b.dat", "w+"); //B Vector.
258     fp_j=fopen("j.dat", "w+"); //Jacobian Matrix.
259     #ifdef READ
260         fp_j2=fopen("j2.dat", "r+"); //Inverse Jacobian Matrix.
261     #else
262         fp_j2=fopen("j2.dat", "w+"); //Inverse Jacobian Matrix.
263     #endif
264     fp_hhp=fopen("hhp.dat", "w+"); //HHP Matrix.
265     fp_hhpPRIME=fopen("hhpPRIME.dat", "w+"); //D(HHP)/D(PHI)
266     Matrix.
267     fp_v=fopen("v.dat", "w+"); //PSI Values (written in a line).
268     fp_vxy=fopen("vxy.dat", "w+"); //PSI Values (written in a grid
269     ).
270     fp_P_HHP=fopen("P_HHP.dat", "w+"); //PSI vs. HHP.
271     fp_change=fopen("change.dat", "w+"); //Matrix of change of PSI
272     between iterations.
273
274     //Exits the program immediately if a file cannot be opened,
275     to prevent unnecessary time wasting.
276     if (NULL==fp_hhp | NULL==fp_r | NULL==fp_z | NULL==fp_v)
277     {
278         printf("Error - Cannot open file.\n");
279         exit(0);
280     }

```

```

277         //Main program, separated into parts to show the user
           progress.
278     printf("\n1\n");
279     initialize(); //Case specifc discretization of the
           differential equation and boundary.
280     printf("2\n");
281     star(); //Case specific insertion of the star.
282     printf("3\n");
283     create_matrix(); //Creates the actual matrix "A" and vector "
           B".
284     printf("4\n");
285     #ifdef READ
286         read(); //Reads J_INVERSE.
287     #endif
288     equationBuilder((N_MAX_X-2)*(N_MAX_Y-2)); //Creates equations
           from "A", "B", and "HHP" and applies Newton's method
           until convergence.
289     printf("5\n");
290
291     _fcloseall();
292 }
293
294 //There are two possible coordinate systems. (x,y) reflects a grid,
           and (s) is just all the grid points in a line.
295 //Converts between (x,y) and (s)
296 __forceinline int RN(int m, int n)
297 {
298     return n*(N_MAX_X-2)+m;
299 }
300
301 //Converts between (s) and (x,y)
302 __forceinline int CN_X(int r)
303 {
304     return r%(N_MAX_X-2);
305 }
306
307 __forceinline int CN_Y(int r)
308 {
309     return r/(N_MAX_X-2);
310 }
311
312 void writeToFiles(void)
313 {
314     rewind(fp_r);
315     rewind(fp_z);
316     rewind(fp_a);
317     rewind(fp_b);
318     rewind(fp_j);
319     rewind(fp_j2);
320     rewind(fp_hhp);
321     rewind(fp_hhpPRIME);
322     rewind(fp_v);

```

```

323     rewind(fp_vxy);
324     rewind(fp_P_HHP);
325     rewind(fp_change);
326
327     //Write actual R values.
328     for (int j=0;j<=N_MAX_X-2;j++)
329     {
330         fprintf(fp_r, "%f\n",j*DX);
331     }
332
333     //Write actual Z values.
334     for (int k=N_MAX_Y-2;k>=0;k--)
335     {
336         fprintf(fp_z, "%f\n",k*DY);
337     }
338
339     //Write "A" matrix .
340     for (int j=0;j<=(N_MAX_X-2)*(N_MAX_Y-2)-1;j++)
341     {
342         for (int k=0;k<=(N_MAX_X-2)*(N_MAX_Y-2)-1;k++)
343         {
344             fprintf(fp_a, "%40.20f ",A[j][k]);
345         }
346         fprintf(fp_a, "\n");
347     }
348
349     //Write "b" vector.
350     for (int k=0;k<(N_MAX_X-2)*(N_MAX_Y-2);k++)
351     {
352         fprintf(fp_b, "%40.20f\n",B[k]);
353     }
354
355     //Write "J" matrix.
356     for (int j=0;j<=(N_MAX_X-2)*(N_MAX_Y-2)-1;j++)
357     {
358         for (int k=0;k<=(N_MAX_X-2)*(N_MAX_Y-2)-1;k++)
359         {
360             fprintf(fp_j, "%40.20f ",J[j][k]);
361         }
362         fprintf(fp_j, "\n");
363     }
364
365     #ifndef READ
366         //Write "J-1" matrix .
367         for (int j=0;j<=(N_MAX_X-2)*(N_MAX_Y-2)-1;j++)
368         {
369             for (int k=0;k<=(N_MAX_X-2)*(N_MAX_Y-2)-1;k++)
370             {
371                 fprintf(fp_j2, "%40.20f ",J_INVERSE[j][k])
372                 ;
373             }
374             fprintf(fp_j2, "\n");

```

```

374     }
375 #endif
376
377     //Write HPP.
378     for (int k=N_MAX_Y-3;k>=0;k--)
379     {
380         for (int j=0;j<N_MAX_X-2;j++)
381         {
382             fprintf(fp_hhp, "%40.20f ",HHP[j][k]);
383         }
384         fprintf(fp_hhp, "\n");
385     }
386
387     //Write D(HHP)/D(PHI).
388     for (int k=N_MAX_Y-3;k>=0;k--)
389     {
390         for (int j=0;j<N_MAX_X-2;j++)
391         {
392             fprintf(fp_hhpPRIME, "%40.20f ",HHP_PRIME[j][
393                 k]);
394         }
395         fprintf(fp_hhpPRIME, "\n");
396     }
397
398     //Write "v" vector.
399     fprintf(fp_v, "%40.20f ",0.0);
400     for (int j=0;j<N_MAX_X-2;j++)
401     {
402         //Deal with bottom boundary.
403         #if TYPE==MONOPOLE
404             //Monopole P=P_OP
405             fprintf(fp_v, "%40.20f ",P_OP);
406         #else
407             //Others
408             if (j<N_LC+1)
409             {
410                 //All non-monopole cases Pz=0
411                 fprintf(fp_v, "%40.20f ",v0[RN(j,0)]);
412             }
413             else
414             {
415                 #if (TYPE==STANDARD||TYPE==JETS)
416                     //CKF, Takamori, Jets P=P_OP
417                     fprintf(fp_v, "%40.20f ",P_OP);
418                 #elif TYPE==NULL
419                     //CKF_NULL, TAK_NULL  $H^2=(R^2-1)*(Pz)^2$ 
420                     fprintf(fp_v, "%40.20f ",v0[RN(j,0)]-L_func[j
421                         ]);
422                 #endif
423             }
424         }
425     }
426 #endif
427 #endif

```

```

424     }
425     for (int k=0;k<N_MAX_Y-2;k++)
426     {
427         fprintf(fp_v, "%40.20f ",0.0);
428         for (int j=0;j<N_MAX_X-2;j++)
429         {
430             fprintf(fp_v, "%40.20f ",v0[RN(j,k)]);
431         }
432     }
433
434     //Write "v" grid.
435     for (int k=N_MAX_Y-3;k>=0;k--)
436     {
437         for (int j=0;j<N_MAX_X-2;j++)
438         {
439             fprintf(fp_vxy, "%40.20f ",v0[RN(j,k)]);
440         }
441         fprintf(fp_vxy, "\n");
442     }
443
444     //Write P vs HHP list.
445     for(int k=0;k<=N_MAX_Y-3;k++)
446     {
447         for(int j=0;j<=N_MAX_X-3;j++)
448         {
449             if(j>N_S_X||k>N_S_Y)
450             {
451                 fprintf(fp_P_HHP,"%40.20f %40.20f\n",
452                     v0[RN(j,k)],HHP[j][k]);
453             }
454         }
455     }
456
457     //Write "change of PSI" grid.
458     for (int k=N_MAX_Y-3;k>=0;k--)
459     {
460         for (int j=0;j<N_MAX_X-2;j++)
461         {
462             fprintf(fp_change, "%40.20f ",CHANGE[j][k]);
463         }
464         fprintf(fp_change, "\n");
465     }
466
467     fflush(fp_r);
468     fflush(fp_z);
469     fflush(fp_a);
470     fflush(fp_b);
471     fflush(fp_j);
472     fflush(fp_j2);
473     fflush(fp_hhp);
474     fflush(fp_hhpPRIME);
475     fflush(fp_v);

```

```

475         fflush(fp_vxy);
476         fflush(fp_P_HHP);
477         fflush(fp_change);
478     }
479
480     //Function to allocate memory for a 2D array of elements of type T.
481     T** Make2DArray(int arraySizeX, int arraySizeY)
482     {
483         T** theArray;
484         theArray = (T**) malloc(arraySizeX*sizeof(T*));
485         for (int i=0;i<arraySizeX;i++)
486         {
487             theArray[i] = (T*) malloc(arraySizeY*sizeof(T));
488         }
489         return theArray;
490     }

```

F.1.2 matrix_build.c

```
1  __forceinline void create_matrix(void)
2  {
3      /"Interior interior" points (points that do not touch a
4      boundary)
5      for(int s=1; s<=N_MAX_X-4; s++)
6      {
7          for(int t=1; t<=N_MAX_Y-4; t++)
8          {
9              A[RN(s,t)][RN(s-1,t)]=a1[s][t];
10             A[RN(s,t)][RN(s+1,t)]=a2[s][t];
11             A[RN(s,t)][RN(s,t-1)]=a3[s][t];
12             A[RN(s,t)][RN(s,t+1)]=a4[s][t];
13             A[RN(s,t)][RN(s,t)]=a5[s][t];
14             B[RN(s,t)]=a6[s][t]-DX*DX*f(s+1,t+1);
15         }
16     }
17     //Left Edge
18     for(int t=1; t<=N_MAX_Y-4; t++)
19     {
20         int s=0;
21
22         //A[RN(s,t)][RN(s-1,t)]=0;
23         A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a1[s][t]*bL2[t]/bL1[t];
24         A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a1[s][t]*bL3[t]/bL1[t];
25         A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a1[s][t]*bL4[t]/bL1[t];
26         A[RN(s,t)][RN(s,t)]=a5[s][t]-a1[s][t]*bL5[t]/bL1[t];
27         B[RN(s,t)]=a6[s][t]-a1[s][t]*bL6[t]/bL1[t]-DX*DX*f(s+1,t+1);
28     }
29
30     //Right Edge
31     for(int t=1; t<=N_MAX_Y-4; t++)
32     {
33         int s=N_MAX_X-3;
34
35         A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a2[s][t]*bR1[t]/bR2[t];
36         //A[RN(s,t)][RN(s+1,t)]=0;
37         A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a2[s][t]*bR3[t]/bR2[t];
38         A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a2[s][t]*bR4[t]/bR2[t];
39         A[RN(s,t)][RN(s,t)]=a5[s][t]-a2[s][t]*bR5[t]/bR2[t];
40         B[RN(s,t)]=a6[s][t]-a2[s][t]*bR6[t]/bR2[t]-DX*DX*f(s+1,t+1);
41     }
```



```

42
43 //Bottom Edge
44 for(int s=1;s<=N_MAX_X-4;s++)
45 {
46     int t=0;
47
48     A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a3[s][t]*bB1[s]/bB3[s];
49     A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a3[s][t]*bB2[s]/bB3[s];
50     //A[RN(s,t)][RN(s,t-1)]=0;
51     A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a3[s][t]*bB4[s]/bB3[s];
52     A[RN(s,t)][RN(s,t)]=a5[s][t]-a3[s][t]*bB5[s]/bB3[s];
53     B[RN(s,t)]=a6[s][t]-a3[s][t]*bB6[s]/bB3[s]-DX*DX*f(s
        +1,t+1);
54 }
55
56 //Top Edge
57 for(int s=1;s<=N_MAX_X-4;s++)
58 {
59     int t=N_MAX_Y-3;
60
61     A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a4[s][t]*bT1[s]/bT4[s];
62     A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a4[s][t]*bT2[s]/bT4[s];
63     A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a4[s][t]*bT3[s]/bT4[s];
64     //A[RN(s,t)][RN(s,t+1)]=0;
65     A[RN(s,t)][RN(s,t)]=a5[s][t]-a4[s][t]*bT5[s]/bT4[s];
66     B[RN(s,t)]=a6[s][t]-a4[s][t]*bT6[s]/bT4[s]-DX*DX*f(s
        +1,t+1);
67 }
68
69 //Bottom Left Corner
70 {
71     int s=0;
72     int t=0;
73
74     //A[RN(s,t)][RN(s-1,t)]=0;
75     A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a1[s][t]*bL2[t]/bL1[t]
        -a3[s][t]*bB2[s]/bB3[s];
76     //A[RN(s,t)][RN(s,t-1)]=0;
77     A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a1[s][t]*bL4[t]/bL1[t]
        -a3[s][t]*bB4[s]/bB3[s];
78     A[RN(s,t)][RN(s,t)]=a5[s][t]-a1[s][t]*bL5[t]/bL1[t]-
        a3[s][t]*bB5[s]/bB3[s];
79     B[RN(s,t)]=a6[s][t]-a1[s][t]*bL6[t]/bL1[t]-a3[s][t]*
        bB6[s]/bB3[s]-DX*DX*f(s+1,t+1);
80 }
81

```

```

82      //Top Left Corner
83      {
84          int s=0;
85          int t=N_MAX_Y-3;
86
87          //A[RN(s,t)][RN(s-1,t)]=0;
88          A[RN(s,t)][RN(s+1,t)]=a2[s][t]-a1[s][t]*bL2[t]/bL1[t]
            -a4[s][t]*bT2[s]/bT4[s];
89          A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a1[s][t]*bL3[t]/bL1[t]
            -a4[s][t]*bT3[s]/bT4[s];
90          //A[RN(s,t)][RN(s,t+1)]=0;
91          A[RN(s,t)][RN(s,t)]=a5[s][t]-a1[s][t]*bL5[t]/bL1[t]-
            a4[s][t]*bT5[s]/bT4[s];
92          B[RN(s,t)]=a6[s][t]-a1[s][t]*bL6[t]/bL1[t]-a4[s][t]*
            bT6[s]/bT4[s]-DX*DX*f(s+1,t+1);
93      }
94
95      //Bottom Right Corner
96      {
97          int s=N_MAX_X-3;
98          int t=0;
99
100         A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a2[s][t]*bR1[t]/bR2[t]
            -a3[s][t]*bB1[s]/bB3[s];
101         //A[RN(s,t)][RN(s+1,t)]=0;
102         //A[RN(s,t)][RN(s,t-1)]=0;
103         A[RN(s,t)][RN(s,t+1)]=a4[s][t]-a2[s][t]*bR4[t]/bR2[t]
            -a3[s][t]*bB4[s]/bB3[s];
104         A[RN(s,t)][RN(s,t)]=a5[s][t]-a2[s][t]*bR5[t]/bR2[t]-
            a3[s][t]*bB5[s]/bB3[s];
105         B[RN(s,t)]=a6[s][t]-a2[s][t]*bR6[t]/bR2[t]-a3[s][t]*
            bB6[s]/bB3[s]-DX*DX*f(s+1,t+1);
106     }
107
108     //Top Right Corner
109     {
110         int s=N_MAX_X-3;
111         int t=N_MAX_Y-3;
112
113         A[RN(s,t)][RN(s-1,t)]=a1[s][t]-a2[s][t]*bR1[t]/bR2[t]
            -a4[s][t]*bT1[s]/bT4[s];
114         //A[RN(s,t)][RN(s+1,t)]=0;
115         A[RN(s,t)][RN(s,t-1)]=a3[s][t]-a2[s][t]*bR3[t]/bR2[t]
            -a4[s][t]*bT3[s]/bT4[s];
116         //A[RN(s,t)][RN(s,t+1)]=0;
117         A[RN(s,t)][RN(s,t)]=a5[s][t]-a2[s][t]*bR5[t]/bR2[t]-
            a4[s][t]*bT5[s]/bT4[s];
118         B[RN(s,t)]=a6[s][t]-a2[s][t]*bR6[t]/bR2[t]-a4[s][t]*
            bT6[s]/bT4[s]-DX*DX*f(s+1,t+1);
119     }
120 }

```

F.1.3 inverse.c

```
1 //Returns the inverse of the matrix stored in 2D array "m"
2 void inverse(T** m,int size,T** g)
3 {
4     //Creates a copy of the input matrix so that the original is
5     //not destroyed
6     T **f;
7     f=Make2DArray( (N_MAX_X-2)*(N_MAX_Y-2), (N_MAX_X-2)*(N_MAX_Y
8     -2));
9     printf("MBytes f=%f\n", (T)SIZE_T_XY_XY/(1024*1024));
10
11     //The memcpy calls won't work if "f" is dynamically allocated
12     //, so we have to manually copy "f" below.
13     //memcpy(f,m, (N_MAX_X-2)*(N_MAX_Y-2)*(N_MAX_X-2)*(N_MAX_Y-2))
14     ;
15
16     //Initializes g as the identity matrix, which will turn into
17     //the inverse of the original matrix
18     for(int x=0;x<size;x++)
19     {
20         for(int y=0;y<size;y++)
21         {
22             g[x][y]=(x==y);
23             f[x][y]=m[x][y];
24         }
25     }
26     for(int j=0;j<size;j++)
27     {
28         printf("j=%i out of %i\n",j+1,size);
29         /*
30         //This code decides when to swap rows, and if the
31         //matrix is singular.
32         //The memcpy calls won't work if "f" is dynamically
33         //allocated.
34         //This needs to be rewritten
35         if(fabs(f[j][j])<0.0000001)
36         {
37             //const char key=_getch();
38             //exit(0);
39             int isSingular=1;
40             //Swap rows to avoid a zero pivot
41             for(int s=j;s<size;s++)
42             {
43                 if(fabs(f[s][j])>0.0000001)
44                 {
45                     T temp[(N_MAX_X-2)*(N_MAX_Y
46                     -2)];
47
48                     isSingular=0;
49                     memcpy(temp,f+j,SIZE_T_XY);
50                     memcpy(f+j,f+s,SIZE_T_XY);
```

```

43         memcpy(f+s,temp,SIZE_T_XY);
44         memcpy(temp,g+j,SIZE_T_XY);
45         memcpy(g+j,g+s,SIZE_T_XY);
46         memcpy(g+s,temp,SIZE_T_XY);
47         break;
48     }
49 }
50 //if(isSingular)
51 {
52     //printf("YOUR JACOBIAN IS SINGULAR
53     //printf("DET_J=%f\n",determinantAlt(
54     //printf("DET_A=%f\n",DET_A);
55     //exit(0);
56     //return;
57 }
58 //multAccum*=-f[j][j];
59 }
60 else
61 {
62     //multAccum*=f[j][j];
63 }
64 */
65 T temp=1/f[j][j];
66 for(int l=size-1;l>j;l--)
67 {
68     f[j][l]*=temp;
69 }
70 for(int l=j;l>=0;l--)
71 {
72     g[j][l]*=temp;
73 }
74
75 for(int k1=0;k1<j;k1++)
76 {
77     for(int k2=size-1;k2>j;k2--)
78     {
79         f[k1][k2]-=f[k1][j]*f[j][k2];
80     }
81     for(int k2=j;k2>=0;k2--)
82     {
83         g[k1][k2]-=f[k1][j]*g[j][k2];
84     }
85 }
86 for(int k1=j+1;k1<size;k1++)
87 {
88     for(int k2=size-1;k2>j;k2--)
89     {
90         f[k1][k2]-=f[k1][j]*f[j][k2];
91     }
92     for(int k2=j;k2>=0;k2--)

```

```

93         {
94             g[k1][k2]-=f[k1][j]*g[j][k2];
95         }
96     }
97 }
98 }
```

F.1.4 equation_solver.c

```
1 //Solves for PHI using Newton's method.
2 //Note: PHI is stored in v0 and v1.
3 void equationBuilder(int size)
4 {
5     T vINIT[ (N_MAX_X-2) * (N_MAX_Y-2) ];
6
7 #ifndef READ
8     for(int x=0;x<size;x++)
9     {
10         for(int y=0;y<size;y++)
11         {
12             J[x][y]=A[x][y];
13             h[x][y]=A[x][y];
14         }
15     }
16 #endif
17
18 #ifdef SHORTCUT
19 #ifndef READ
20     printf("pre inverse\n");
21     inverse(J,size,J_INVERSE);
22     printf("after inverse\n");
23 #endif
24 #endif
25
26     //Our initial guess for every point is one.
27     for(int i=0;i<size;i++)
28     {
29         vINIT[i]=1.0;
30         v0[i]=vINIT[i];
31         v1[i]=vINIT[i];
32     }
33
34     /*
35     //An alternate way of setting initial values.
36     //Our initial guess for every point is "random" between 0 and
37     1.
38     srand(time(NULL));
39     for(int i=0;i<size;i++)
40     {
41         vINIT[i]=( (T) rand() ) / RAND_MAX;
42         v0[i]=vINIT[i];
43         v1[i]=vINIT[i];
44     }
45     */
46
47     T olddeltaALL=0;
48     T deltaALL=0;
49     for(int iiii=1;iiii<=NUM_ITERATIONS_MAX;iiii++)
50     {
```

```

50      olddeltaALL=deltaALL;
51      deltaALL=0;
52      T deltaMAX=0;
53
54      //This code controls user interaction. All user input
           is direct keyboard hits, NOT typing text into a
           command line.
55      //To the best of my knowledge, this code to check for
           user input does NOT slow the program down
           significantly.
56      if(_kbhit())
57      {
58          //This call to _getch() will not pause the
           program because the user already hit a key
           ,
59          //but in general the other calls will pause
           the program and wait for input.
60          const char key=_getch();
61          if (key=='b' || key=='B')
62          {
63              //Simulation is paused.
64              printf("\nWriting data to files
           ..... \n");
65              writeToFiles();
66              printf("\nData has been written to
           files.\n\nEnter 'w' or 'W' to
           change the FRAC parameter.\nEnter
           'e' or 'E' to change BETA.\nEnter
           'r' or 'R' to reset the same
           simulation.\nEnter 'c' or 'C' to
           reset with another BETA value.\n
           Enter 'q' or 'Q' to quit.\nEnter
           anything else to continue.\n");
67
68              const char key=_getch();
69              if (key=='w' || key=='W')
70              {
71                  //Yes, the code needs to be
           structured this way!
72                  while(1)
73                  {
74                      printf("Enter new
           V_FRAC value:\n");
75                      scanf("%lf",&V_FRAC);
76                      fflush(stdin); //NOT
           STANDARD :(
77                      rewind(stdin); //NOT
           STANDARD :(
78                      printf("\nV_FRAC=%lf
           - Is this OK?\n
           Enter 'y' or 'Y'
           to continue\nEnter

```

```

        'a' or 'A' to try
        again.\n\n",
        V_FRAC);

79
80     char key;
81     while(1)
82     {
83         key=_getch();
84         if (key=='y'
            ||key=='Y'
            ||key=='a'
            ||key=='A'
            )
            {
85                 break
86                 ;
87             }
88     }
89     if (key=='y' ||key=='Y'
        ')
        {
90             break;
91         }
92     }
93 }
94 }
95 else if (key=='e' ||key=='E')
96 {
97     while(1)
98     {
99         printf("Enter new
        BETA value:\n");
100        scanf("%lf",&BETA);
101        fflush(stdin); //NOT
        STANDARD :(
102        rewind(stdin); //NOT
        STANDARD :(
103        printf("\nBETA=%lf -
        Is this OK?\nEnter
        'y' or 'Y' to
        continue\nEnter 'a
        ' or 'A' to try
        again.\n\n",BETA);

104
105     char key;
106     while(1)
107     {
108         key=_getch();
109         if (key=='y'
            ||key=='Y'
            ||key=='a'
            ||key=='A'
            )

```



```

110                                     {
111                                     break
                                     ;
112                                     }
113                                     }
114                                     if (key=='y' || key=='Y
                                     ')
115                                     {
116                                     break;
117                                     }
118                                     }
119                                     }
120 else if (key=='r' || key=='R')
121 {
122     //Completely reset simulation
123     for(int i=0;i<size;i++)
124     {
125         vINIT[i]=1;
126         v0[i]=vINIT[i];
127         v1[i]=vINIT[i];
128         iiii=1;
129     }
130 }
131 else if (key=='c' || key=='C')
132 {
133     //Completely reset simulation
134     //with different BETA value
135     for(int i=0;i<size;i++)
136     {
137         vINIT[i]=1;
138         v0[i]=vINIT[i];
139         v1[i]=vINIT[i];
140         iiii=1;
141     }
142     while(1)
143     {
144         printf("Enter new
145             BETA value:\n");
146         scanf("%lf",&BETA);
147         fflush(stdin); //NOT
148             STANDARD :(
149         rewind(stdin); //NOT
150             STANDARD :(
151         printf("\nBETA=%lf -
152             Is this OK?\nEnter
153             'y' or 'Y' to
154             continue\nEnter 'a
155             ' or 'A' to try
156             again.\n\n",BETA);

```

```

149
150         char key;
151         while(1)
152         {
153             key=_getch();
154             if (key=='y'
                ||key=='Y'
                ||key=='a'
                ||key=='A'
                )
            {
155                 break
156                 ;
            }
157         }
158     }
159     if (key=='y' ||key=='Y'
        ')
    {
160         break;
161     }
162 }
163 }
164 }
165 else if (key=='q' ||key=='Q')
166 {
167     //Quits the program.
168     printf(" * * * * * * * * * * *
        * * * * * * * * * * *
        * * * * * * * * * * \n");
169     printf("%i out of %i
        iterations were completed
        .\n",iiii,
        NUM_ITERATIONS_MAX);
170     _fcloseall();
171     exit(0);
172 }
173 else
174 {
175     printf("\n\nCONTINUE\n\n");
176 }
177 }
178 }
179
180 printf("\n          ITERATION=%i      V_FRAC=%f
        BETA=%f\n",iiii,V_FRAC,BETA);
181
182 #ifndef SHORTCUT
183     //Updates the Jacobian for each iteration of Newton's
        Method.
184     for(int x=0;x<size;x++)
185     {
186         p[x]=DX*DX*HHP_PRIME[CN_X(x)][CN_Y(x)];
187         J[x][x]=h[x][x]+p[x];

```

```

188     }
189
190     inverse(J,size,J_INVERSE);
191 #endif
192     //Calculates the nonlinear equations that we are
        trying to solve to be zero.
193     for(int i=0;i<size;i++)
194     {
195         eq[i]=0;
196         for(int k=0;k<size;k++)
197         {
198             eq[i]+=A[i][k]*v0[k];
199         }
200         eq[i]+=-B[i]+HHP[CN_X(i)][CN_Y(i)]*DX*DX;
201     }
202
203     int s_max=0;
204     //Newton's method.
205     for(int i=0;i<size;i++)
206     {
207         //We could do better by not including the
            star in our solver, since the values are
            already known.
208         //if((CN_X(i)>(N_S_X-1))||(CN_Y(i)>(N_S_Y-1))
            )
209         {
210             //Update all variables.
211             v1[i]=0;
212             for(int j=0;j<size;j++)
213             {
214                 v1[i]+=J_INVERSE[i][j]*eq[j];
215             }
216             v1[i]=v0[i]-v1[i];
217
218             //Keeps track of changes between
                iterations.
219             deltaALL+=fabs(v0[i]-v1[i]);
220             CHANGE[CN_X(i)][CN_Y(i)]=fabs(v0[i]-
                v1[i]);
221             if((CN_X(i)>(N_S_X-1))||(CN_Y(i)>(
                N_S_Y-1)))
222             {
223                 if (fabs(v0[i]-v1[i])>
                    deltaMAX)
224                 {
225                     s_max=i;
226                     deltaMAX=fabs(v0[i]-
                        v1[i]);
227                 }
228             }
229         }
230     }

```

```

231     printf("Max change at %i, or (%i,%i)\n",s_max,CN_X(
        s_max),CN_Y(s_max));
232     printf("deltaALL=%f\n",deltaALL);
233     printf("olddeltaALL=%f\n",olddeltaALL);
234     printf("deltaMAX=%f\n",deltaMAX);
235
236     //Similar effect to the relaxation parameter.
237     for(int i=0;i<size;i++)
238     {
239         v0[i]=(V_FRAC)*(v1[i])+(1-V_FRAC)*(v0[i]);
240     }
241
242     //Automatically end if a certain convergence level is
        reached.
243     //User is provided with options as to what to do next
244
245     if(iiii>=5&&deltaMAX<0.000001)
246     {
247         printf("STABLE BREAK\n");
248         printf("\nWriting data to files.....\n
        ");
249         writeToFiles();
250
251         printf("\nData has been written to files.\n\
        nEnter 'r' or 'R' to reset the same
        simulation.\nEnter 'c' or 'C' to reset
        with another BETA value.\nEnter 'e' or 'E'
        to continue with another BETA value.\
        nEnter anything else to quit.\n");
252         const char key=_getch();
253
254         if (key=='r' ||key=='R')
255         {
256             //Completely reset simulation.
257             for(int i=0;i<size;i++)
258             {
259                 vINIT[i]=1;
260                 v0[i]=vINIT[i];
261                 v1[i]=vINIT[i];
262             }
263             iiii=0;
264         }
265         else if (key=='c' ||key=='C')
266         {
267             //Completely reset simulation with
                different BETA value.
268             for(int i=0;i<size;i++)
269             {
270                 vINIT[i]=1;
271                 v0[i]=vINIT[i];
272                 v1[i]=vINIT[i];

```

```

273      iiii=0;
274
275      while(1)
276      {
277          printf("Enter new BETA value
                :\n");
278          scanf("%lf",&BETA);
279          fflush(stdin);//NOT STANDARD
                : (
280          rewind(stdin);//NOT STANDARD
                : (
281          printf("\nBETA=%lf - Is this
                OK?\nEnter 'y' or 'Y' to
                continue\nEnter 'a' or 'A'
                to try again.\n\n",BETA);
282
283          char key;
284          while(1)
285          {
286              key=_getch();
287              if (key=='y' || key=='Y'
                ' || key=='a' || key=='A')
                {
288                  break;
289              }
290          }
291          if (key=='y' || key=='Y')
292          {
293              break;
294          }
295      }
296  }
297  }
298  else if (key=='e' || key=='E')
299  {
300      //Continue simulation with different
                BETA value.
301      iiii=0; //This is reset here so that
                the maximum iteration limit isn't
                reached prematurely if you switch
                BETA a lot.
302      while(1)
303      {
304          printf("Enter new BETA value
                :\n");
305          scanf("%lf",&BETA);
306          fflush(stdin);//NOT STANDARD
                : (
307          rewind(stdin);//NOT STANDARD
                : (
308          printf("\nBETA=%lf - Is this
                OK?\nEnter 'y' or 'Y' to

```

```

309         continue\nEnter 'a' or 'A'
310         to try again.\n\n",BETA);
311
312         char key;
313         while(1)
314         {
315             key=_getch();
316             if (key=='y' ||key=='Y'
317                 ' ||key=='a' ||key=='A')
318             {
319                 break;
320             }
321         }
322         if (key=='y' ||key=='Y')
323         {
324             break;
325         }
326     }
327     else
328     {
329         //Quits the program.
330         printf(" * * * * * \n");
331         printf("%i out of %i iterations were
332             completed.\n",iiii,
333             NUM_ITERATIONS_MAX);
334         _fcloseall();
335         exit(0);
336     }
337 }
338 }

```

335 //Update HHP values.
336 hhpSet (HHP,HHP_PRIME);

F.1.5 equation_solver_2case.c

```
1 //Solves for PHI using Newton's method.
2 //Note: PHI is stored in v0 and v1.
3 void equationBuilder(int size)
4 {
5     T vINIT[ (N_MAX_X-2) * (N_MAX_Y-2) ];
6
7 #ifndef READ
8     for(int x=0;x<size;x++)
9     {
10         for(int y=0;y<size;y++)
11         {
12             J[x][y]=A[x][y];
13             h[x][y]=A[x][y];
14         }
15     }
16 #endif
17
18 #ifdef SHORTCUT
19 #ifndef READ
20     printf("pre inverse\n");
21     inverse(J,size,J_INVERSE);
22     printf("after inverse\n");
23 #endif
24 #endif
25
26     //Our initial guess for every point is one
27     for(int i=0;i<size;i++)
28     {
29         vINIT[i]=1.0;
30         v0[i]=vINIT[i];
31         v1[i]=vINIT[i];
32     }
33
34     /*
35     //An alternate way of setting initial values.
36     //Our initial guess for every point is "random" between 0 and
37     1.
38     srand(time(NULL));
39     for(int i=0;i<size;i++)
40     {
41         vINIT[i]=( (T) rand() ) / RAND_MAX;
42         v0[i]=vINIT[i];
43         v1[i]=vINIT[i];
44     }
45     */
46
47     T olddeltaALL=0;
48     T deltaALL=0;
49     for(int iiii=1;iiii<=NUM_ITERATIONS_MAX;iiii++)
50     {
```

```

50      olddeltaALL=deltaALL;
51      deltaALL=0;
52      T deltaMAX=0;
53
54      //This code controls user interaction. All user input
        is direct keyboard hits, NOT typing text into a
        command line.
55      //To the best of my knowledge, this code to check for
        user input does NOT slow the program down
        significantly.
56      if(_kbhit())
57      {
58          //This call to _getch() will not pause the
            program because the user already hit a key
            ,
59          //but in general the other calls will pause
            the program and wait for input.
60          const char key=_getch();
61          if (key=='b' || key=='B')
62          {
63              //Simulation is paused.
64              printf("\nWriting data to files
                ..... \n");
65              writeToFiles();
66              printf("\nData has been written to
                files.\n\nEnter 'w' or 'W' to
                change the FRAC parameter.\nEnter
                'q' or 'Q' to quit.\nEnter
                anything else to continue.\n");
67
68              const char key=_getch();
69              if (key=='w' || key=='W')
70              {
71                  //Yes, the code needs to be
                    structured this way!
72                  while(1)
73                  {
74                      printf("Enter new
                        V_FRAC value:\n");
75                      scanf("%lf",&V_FRAC);
76                      fflush(stdin); //NOT
                        STANDARD :(
77                      rewind(stdin); //NOT
                        STANDARD :(
78                      printf("\nV_FRAC=%lf
                        - Is this OK?\n
                        nEnter 'y' or 'Y'
                        to continue\nEnter
                        'a' or 'A' to try
                        again.\n\n",
                        V_FRAC);
79

```



```

80         char key;
81         while(1)
82         {
83             key=_getch();
84             if (key=='y'
                ||key=='Y'
                ||key=='a'
                ||key=='A'
                )
            {
85                 break
86                 ;
87             }
88         }
89         if (key=='y' ||key=='Y'
            ' )
90         {
91             break;
92         }
93     }
94 }
95 else if (key=='q' ||key=='Q')
96 {
97     //If you are at the first
        case of a double
        simulation, it switches to
        the second case.
98     //Otherwise, it quits the
        program.
99     if(!toggle)
100     {
101         printf("\n\n\
            nSwitching Case\n\
            n\n");
102         toggle=1;
103         iiii=0;
104         for(int i=0;i<size;i
            ++ )
105         {
106             vINIT[i]=1.0;
107             v0[i]=vINIT[i
                ];
108             v1[i]=vINIT[i
                ];
109         }
110     }
111     else
112     {
113         printf(" * * * * * * *
            * * * * * * * * *
            * * * * * * * * *
            * * * * * * * *

```

```

114                                     *\n");
                                     printf("%i out of %i
                                     iterations were
                                     completed.\n",iiii
                                     ,
                                     NUM_ITERATIONS_MAX
                                     );
115                                     _fcloseall();
116                                     exit(0);
117                                     }
118                                     }
119                                     else
120                                     {
121                                     printf("\n\nCONTINUE\n\n");
122                                     }
123                                     }
124                                     }
125
126                                     printf("\n          ITERATION=%i          V_FRAC=%f\n",iiii,
                                     V_FRAC);
127
128 #ifndef SHORTCUT
129                                     //Updates the Jacobian for each iteration of Newton's
                                     Method.
130                                     for(int x=0;x<size;x++)
131                                     {
132                                     p[x]=DX*DX*HHP_PRIME[CN_X(x)][CN_Y(x)];
133                                     J[x][x]=h[x][x]+p[x];
134                                     }
135
136                                     inverse(J,size,J_INVERSE);
137 #endif
138                                     //Calculates the nonlinear equations that we are
                                     trying to solve to be zero.
139                                     for(int i=0;i<size;i++)
140                                     {
141                                     eq[i]=0;
142                                     for(int k=0;k<size;k++)
143                                     {
144                                     eq[i]+=A[i][k]*v0[k];
145                                     }
146                                     eq[i]+=-B[i]+HHP[CN_X(i)][CN_Y(i)]*DX*DX;
147                                     }
148
149                                     int s_max=0;
150                                     //Newton's method.
151                                     for(int i=0;i<size;i++)
152                                     {
153                                     //We could do better by not including the
                                     star in our solver, since the values are
                                     already known.
154                                     //if((CN_X(i)>(N_S-1))||(CN_Y(i)>(N_S-1)))

```

```

155         {
156             //Update all variables.
157             v1[i]=0;
158             for(int j=0;j<size;j++)
159             {
160                 v1[i]+=J_INVERSE[i][j]*eq[j];
161             }
162             v1[i]=v0[i]-v1[i];
163
164             //Keeps track of changes between
165             //iterations.
166             deltaALL+=fabs(v0[i]-v1[i]);
167             CHANGE[CN_X(i)][CN_Y(i)]=fabs(v0[i]-
168             v1[i]);
169             if((CN_X(i)>(N_S_X-1))||(CN_Y(i)>(
170             N_S_Y-1)))
171             {
172                 if(fabs(v0[i]-v1[i])>
173                 deltaMAX)
174                 {
175                     s_max=i;
176                     deltaMAX=fabs(v0[i]-
177                     v1[i]);
178                 }
179             }
180         }
181     }
182     printf("Max change at %i, or (%i,%i)\n",s_max,CN_X(
183     s_max),CN_Y(s_max));
184     printf("deltaALL=%f\n",deltaALL);
185     printf("olddeltaALL=%f\n",olddeltaALL);
186     printf("deltaMAX=%f\n",deltaMAX);
187
188     //Similar effect to the relaxation parameter.
189     for(int i=0;i<size;i++)
190     {
191         v0[i]=(V_FRAC)*(v1[i])+(1-V_FRAC)*(v0[i]);
192     }
193
194     //Automatically end/toggle if a certain convergence
195     //level is reached.
196     if(iiii>=5&&deltaMAX<0.000001)
197     {
198         if(!toggle)
199         {
200             printf("\n\n\nSwitching Case\n\n\n");
201             toggle=1;
202             iiii=0;
203             for(int i=0;i<size;i++)
204             {
205                 vINIT[i]=1.0;
206                 v0[i]=vINIT[i];

```

```

200                                     v1[i]=vINIT[i];
201                                     }
202                                 }
203                                 else
204                                 {
205                                     printf("STABLE BREAK\n");
206                                     printf("\nWriting data to files
                                     ..... \n");
207                                     writeToFiles();
208                                     break;
209                                 }
210                             }
211
212                             //Update HHP values.
213                             hhpSet (HHP,HHP_PRIME);
214                         }
215 }

```

F.1.6 reset_ckf.c

```
1 void InsertionSort(void)
2 {
3     //This is the common "insertion sort" algorithm for sorting
4     an array. Here, we sort L_X in descending order, and carry
5     L_Y "along for the ride."
6     //This means that L_X vs. L_Y pairs are preserved. We are
7     simply ordering the pairs based on L_X values.
8     int i, j;
9     T tempX, tempY;
10    for (j=1; j<(N_MAX_X-2)*(N_MAX_Y-2); j++) // Start with 1, not 0,
11    because first value in any array is "automatically"
12    smaller than everything before it.
13    {
14        //At every loop iteration, we consider the value L_X[
15        j]. Everything before this value is already in
16        descending order.
17        tempX = L_X[j]; //Store current value in consideration
18        tempY = L_Y[j]; //All sorting is done with L_X values
19        only, not L_Y.
20        for (i=j-1; (i>=0)&&(L_X[i]<tempX); i--) //Values smaller
21        than tempX rise, and tempX "sinks" until it hits
22        a value greater than itself.
23        {
24            L_X[i+1] = L_X[i];
25            L_Y[i+1] = L_Y[i];
26        }
27        L_X[i+1] = tempX;
28        L_Y[i+1] = tempY;
29    }
30    return;
31 }
32
33 void resetCKF(void)
34 {
35     int size=(N_MAX_X-2)*(N_MAX_Y-2);
36     //We need to get PSI vs. HHP so we can take integral via
37     trapezoidal method.
38     for (int i=0; i<size; i++)
39     {
40         if (CN_Y(i)>0)
41         {
42             L_X[i]=v0[i];
43             L_Y[i]=fabs(HHP[CN_X(i)][CN_Y(i)]); //Do we
44             really need to take the absolute value.
45         }
46     }
47     InsertionSort(); //This makes L_X sorted in DESCENDING order
48     (L_X[0] is largest, L_X[size-1] is smallest).
49 }
```

```

37 //Bottom Edge (past light cylinder).
38 for(int s=N_LC;s<=N_MAX_X-4;s++)
39 {
40     T sum=0;
41     if(v0[s]<P_OP)
42     {
43         for(int i=size-1;i>0;i--)
44         {
45             if(L_X[i]>v0[s]){break;}
46             sum+=0.5*(L_X[i-1]-L_X[i])*(L_Y[i]+
47                 L_Y[i-1]);
48         }
49         sum=fabs(sum);//This ensures that the term
50         under the square root is positive. We
51         shouldn't really need this...
52     }
53
54     //Note: H is negative, so we want -sqrt(H^2).
55     bB1[s]=0;
56     bB2[s]=0;
57     bB3[s]=-1;
58     bB4[s]=0;
59     bB5[s]=1;
60     bB6[s]=-DY*sqrt(2*sum/((s+1)*(s+1)*DX*DX-1));
61
62     L_func[s]=-DY*sqrt(2*sum/((s+1)*(s+1)*DX*DX-1)); //
63     This is used just so that we can add the bottom
64     boundary to plots.
65 }
66
67 //Bottom Right Corner.
68 int s=N_MAX_X-3;
69 T sum=0;
70 if(v0[s]<P_OP)
71 {
72     for(int i=size-1;i>0;i--)
73     {
74         if(L_X[i]>v0[s]){break;}
75         sum+=0.5*(L_X[i-1]-L_X[i])*(L_Y[i]+L_Y[i-1]);
76     }
77 }
78
79 int t=0;
80 int i=s+1;
81 int j=t+1;
82
83 //Note: H is negative, so we want -sqrt(H^2).
84 bR1[t]=-i;
85 bR2[t]=i;
86 bR3[t]=0;
87 bR4[t]=j;
88 bR5[t]=-j;

```

```

84         bR6[t]=DY*j*sqrt(2*sum/(i*i*DX*DX-1));
85
86         bB1[s]=0;
87         bB2[s]=0;
88         bB3[s]=-1;
89         bB4[s]=0;
90         bB5[s]=1;
91         bB6[s]=-DY*sqrt(2*sum/(i*i*DX*DX-1));
92
93         L_func[s]=-DY*sqrt(2*sum/((s+1)*(s+1)*DX*DX-1));
94
95         create_matrix();
96
97         for(int x=0;x<size;x++)
98         {
99             for(int y=0;y<size;y++)
100             {
101                 J[x][y]=A[x][y];
102                 h[x][y]=A[x][y];
103             }
104         }
105     }

```

F.1.7 reset_tak.c

```
1 void resetTak(void)
2 {
3     T size=(N_MAX_X-2)*(N_MAX_Y-2);
4
5     //Bottom Edge (past light cylinder).
6     for(int s=N_LC; s<=N_MAX_X-4; s++)
7     {
8         int t=0;
9         //Note: H is negative, so we want -sqrt(H^2).
10        if(v0[RN(s+1,t+1)]>P_OP){v0[RN(s+1,t+1)]=P_OP;}
11        bB1[s]=0;
12        bB2[s]=0;
13        bB3[s]=-1;
14        bB4[s]=0;
15        bB5[s]=1;
16        bB6[s]=-DY*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)]/
            P_OP)*(pow(fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/sqrt
            ((s+1)*(s+1)*DX*DX-1);
17
18        //This is used just so that we can add the bottom
            boundary to plots.
19        L_func[s]=-DY*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)]/
            P_OP)*(pow(fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/
            sqrt((s+1)*(s+1)*DX*DX-1);
20    }
21
22    //Bottom Right Corner.
23    int s=N_MAX_X-3;
24    int t=0;
25    int i=s+1;
26    int j=t+1;
27
28    //Note: H is negative, so we want -sqrt(H^2).
29    if(v0[RN(s+1,t+1)]>P_OP){v0[RN(s+1,t+1)]=P_OP;}
30    bR1[t]=-i;
31    bR2[t]=i;
32    bR3[t]=0;
33    bR4[t]=j;
34    bR5[t]=-j;
35    bR6[t]=DY*j*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)]/P_OP)*(
        pow(fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/sqrt((s+1)*(s+1)*DX
        *DX-1);
36
37    bB1[s]=0;
38    bB2[s]=0;
39    bB3[s]=-1;
40    bB4[s]=0;
41    bB5[s]=1;
42    bB6[s]=-DY*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)]/P_OP)*(pow
        (fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/sqrt((s+1)*(s+1)*DX*DX
```



```

-1);
43
44 //This is used just so that we can add the bottom boundary to
plots.
45 L_func[s]=-DY*1.07*v0[RN(s+1,t+1)]*(2-v0[RN(s+1,t+1)]/P_OP)*(
pow(fabs(1-v0[RN(s+1,t+1)]/P_OP),0.4))/sqrt((s+1)*(s+1)*DX
*DX-1);
46
47 create_matrix();
48
49 for(int x=0;x<size;x++)
50 {
51     for(int y=0;y<size;y++)
52     {
53         J[x][y]=A[x][y];
54         h[x][y]=A[x][y];
55     }
56 }
57 }

```

F.1.8 read_matrix.c

```
1 void read(void)
2 {
3     rewind(fp_j2);
4
5     int size=(N_MAX_X-2)*(N_MAX_Y-2);
6     for (int j=0;j<=(N_MAX_X-2)*(N_MAX_Y-2)-1;j++)
7     {
8         printf("%i out of %i\n",j+1,size);
9         for (int k=0;k<=(N_MAX_X-2)*(N_MAX_Y-2)-1;k++)
10        {
11            fscanf(fp_j2, "%lf ",&J_INVERSE[j][k]);
12        }
13    }
14
15    fclose(fp_j2);
16 }
```

F.2 Case Files

This section includes the source code for each case file. Every case file is structured the same way, with the discretization choices first, then the boundary conditions, then the choice of star, and finally the method for setting $F(\psi)$ and any other nonhomogeneous terms. Our hope in including these case files is that readers can understand the particular choices made in crafting the equations and the boundaries, along with the subtle decisions that a simulator must make. We hope that by providing every line of code we will answer any questions about the work done and will aid those who wish to pursue the work further.

Single simulation cases:

- `ckf_monopole.c` - Pure Monopole case with the CKF method.
- `ckf.c` - Pure CKF case from Contopoulos et al. 1999.
- `ckf_jets.c` - CKF case mixed with Jets case.
- `ckf_null.c` - Null Sheet case with the CKF method.
- `tak_monopole_test.c` - Test of the Monopole case using the known answer.
- `tak_monopole.c` - Pure Monopole case with the TOTS method.
- `tak_monopole_jets.c` - Monopole case mixed with Jets case.
- `tak.c` - Pure TOTS case from Takamori et al. 2012.
- `tak_jets.c` - TOTS case mixed with Jets case.
- `tak_theory_jets.c` - Theoretical Jets case.
- `tak_null.c` - Null Sheet case with the TOTS method.

Double simulation cases:

- `ckf_monopole_jets.c` - CKF Monopole, then Jets.
- `ckf_tak.c` - CKF, then TOTS.
- `ckf_null_tak.c` - CKF Null Sheet, then TOTS.
- `tak_ckf.c` - TOTS, then CKF.
- `tak_ckf_jets.c` - TOTS, then CKF Jets.
- `tak_ckf_null.c` - TOTS, then CKF Null Sheet.
- `tak_theory_jets_ckf.c` - TOTS Theoretical Jets case, then CKF.

F.2.1 ckf_monopole.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE MONOPOLE
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is backward difference
34             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
35             a2[s][t]=1-i*i*DX*DX;
36             a3[s][t]=1-i*i*DX*DX;
37             a4[s][t]=1-i*i*DX*DX;
38             a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
39             a6[s][t]=0;
40             */
41             //1st derivative is backward difference, DX
42             and DY independent
43             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
44             a2[s][t]=1-i*i*DX*DX;
45             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
46             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```

44         a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
45             -1/i-i*DX*DX;
46         a6[s][t]=0;
47         //Value near LC is average of values to the
48             left and right.
49         if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
50         {
51             a1[s][t]=-0.5;
52             a2[s][t]=-0.5;
53             a3[s][t]=0;
54             a4[s][t]=0;
55             a5[s][t]=1;
56             a6[s][t]=0;
57         }
58         /*
59         //Unused
60         //Polynomial coefficients of HHP (constant
61             term taken care of in function "f")
62         //c[#] is the coefficient for (Pij)^(#+1)
63         c[0][s][t]=4;
64         c[1][s][t]=-6;
65         c[2][s][t]=2;
66         */
67     }
68
69     //Left Boundary P=0
70     for (int t=1;t<=N_MAX_Y-4;t++)
71     {
72         int s=0;
73
74         bL1[t]=1;
75         bL2[t]=0;
76         bL3[t]=0;
77         bL4[t]=0;
78         bL5[t]=0;
79         bL6[t]=0;
80     }
81
82     //Right Boundary RPr+ZPz=0
83     for (int t=1;t<=N_MAX_Y-4;t++)
84     {
85         int s=N_MAX_X-3;
86
87         bR1[t]=-i;
88         bR2[t]=i;
89         bR3[t]=-j;
90         bR4[t]=j;
91         bR5[t]=0;
92         bR6[t]=0;

```

```

93     }
94
95     //Bottom Boundary P=P_OP
96     for(int s=1;s<=N_MAX_X-4;s++)
97     {
98         int t=0;
99
100         bB1[s]=0;
101         bB2[s]=0;
102         bB3[s]=1;
103         bB4[s]=0;
104         bB5[s]=0;
105         bB6[s]=P_OP;
106     }
107
108     //Top Boundary RPr+ZPz=0
109     for(int s=1;s<=N_MAX_X-4;s++)
110     {
111         int t=N_MAX_Y-3;
112
113         bT1[s]=-i;
114         bT2[s]=i;
115         bT3[s]=-j;
116         bT4[s]=j;
117         bT5[s]=0;
118         bT6[s]=0;
119     }
120
121     //Bottom Left Corner
122     {
123         int s=0;
124         int t=0;
125
126         bL1[t]=1;
127         bL2[t]=0;
128         bL3[t]=0;
129         bL4[t]=0;
130         bL5[t]=0;
131         bL6[t]=0;
132
133         bB1[s]=0;
134         bB2[s]=0;
135         bB3[s]=1;
136         bB4[s]=0;
137         bB5[s]=0;
138         bB6[s]=P_OP;
139     }
140
141     //Top Left Corner
142     {
143         int s=0;
144         int t=N_MAX_Y-3;

```

```

145
146         bL1[t]=1;
147         bL2[t]=0;
148         bL3[t]=0;
149         bL4[t]=0;
150         bL5[t]=0;
151         bL6[t]=0;
152
153         bT1[s]=0;
154         bT2[s]=i;
155         bT3[s]=-j;
156         bT4[s]=j;
157         bT5[s]=0;
158         bT6[s]=0;
159     }
160
161     //Bottom Right Corner
162     {
163         int s=N_MAX_X-3;
164         int t=0;
165
166         bR1[t]=-i;
167         bR2[t]=i;
168         bR3[t]=0;
169         bR4[t]=j;
170         bR5[t]=0;
171         bR6[t]=j*P_OP;
172
173         bB1[s]=0;
174         bB2[s]=0;
175         bB3[s]=1;
176         bB4[s]=0;
177         bB5[s]=0;
178         bB6[s]=P_OP;
179     }
180
181     //Top Right Corner
182     {
183         int s=N_MAX_X-3;
184         int t=N_MAX_Y-3;
185
186         bR1[t]=-1;
187         bR2[t]=1;
188         bR3[t]=0;
189         bR4[t]=0;
190         bR5[t]=0;
191         bR6[t]=0;
192
193         bT1[s]=0;
194         bT2[s]=0;
195         bT3[s]=-1;
196         bT4[s]=1;

```

```

197             bT5[s]=0;
198             bT6[s]=0;
199         }
200     }
201
202     //This function inserts the star.
203     void star(void){}
204
205     //This fills in HHP and d(HHP)/d(Psi), which is a function of "Psi"
and may or may not include both linear and nonlinear terms.
206     __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
207     {
208         T HHP_L[N_MAX_Y-2];
209         for(int KP=0;KP<N_MAX_Y-2;KP++)
210         {
211             HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(
                N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
212         }
213
214         //At r<rL, there are many possible cases (closed field lines,
open field lines that do or do not cross the light
cylinder).
215         //For 0<r<=NR_S, we stay OUTSIDE star, because the star value
is known and does not need to be altered.
216
217         for(int jj=0;jj<=N_LC-2;jj++)
218         {
219             for(int kk=0;kk<=N_MAX_Y-3;kk++)
220             {
221                 int KP=0;
222                 {
223                     const T PT=v0[RN(jj,kk)];
224
225                     if (PT<v0[RN(N_LC-2,0)])
226                     {
227                         if (PT<v0[RN(N_LC-2,N_MAX_Y-3)
228                             ])
229                         {
230                             HHP[jj][kk]=HHP_L[
231                                 N_MAX_Y-3]*PT/v0[
232                                     RN(N_LC-2,N_MAX_Y
233                                         -3)];
234
235                             }
236                             else
237                             {
238                                 for (;KP<N_MAX_Y-4;&&PT
239                                     <v0[RN(N_LC-2,KP)
240                                         ];KP++);
241
242                                 const T Q1=PT-v0[RN(
243                                     N_LC-1,KP+1)];

```



```

236                                     const T Q2=PT-v0 [RN (
237                                     N_LC-1, KP) ] ;
                                     HHP [jj] [kk] = (Q1*HHP_L
                                     [KP]-Q2*HHP_L [KP
                                     +1]) / (Q1-Q2);
238                                     }
239                                     }
240                                     else
241                                     {
242                                     HHP [jj] [kk]=0;
243                                     }
244                                     }
245                                     if (isnan (HHP [jj] [kk])) {HHP [jj] [kk]=0;}
246                                     }
247                                     }
248
249 for (int jj=N_LC-1; jj==N_LC-1; jj++)
250 {
251     for (int kk=0; kk<=N_MAX_Y-3; kk++)
252     {
253         HHP [N_LC-1] [kk]=HHP_L [kk];
254     }
255 }
256
257 for (int jj=N_LC; jj<=N_MAX_X-3; jj++)
258 {
259     for (int kk=0; kk<=N_MAX_Y-3; kk++)
260     {
261         int KP=0;
262         const T PT=v0 [RN (jj, kk) ] ;
263
264         for (; KP<N_MAX_Y-4&&PT<v0 [RN (N_LC-2, KP) ] ; KP++)
265             ;
266
267         const T Q1=PT-v0 [RN (N_LC-1, KP+1) ] ;
268         const T Q2=PT-v0 [RN (N_LC-1, KP) ] ;
269         HHP [jj] [kk] = (Q1*HHP_L [KP]-Q2*HHP_L [KP+1]) / (Q1
270             -Q2);
271         if (isnan (HHP [jj] [kk])) {HHP [jj] [kk]=0;}
272     }
273 }
274
275 for (int jj=N_LC-1; jj==N_LC-1; jj++)
276 {
277     for (int kk=0; kk<=N_MAX_Y-3; kk++)
278     {
279         v0 [RN (N_LC-1, kk) ] =0.5* (v0 [RN (N_LC-2, kk) ] +v0 [
280             RN (N_LC, kk) ] ) ;

```

```

281      //TODO - HHP_PRIME is largely unnecessary, and this
282      calculation may be wrong.
283      for(int x=0;x<N_MAX_X-2;x++)
284      {
285          for(int y=0;y<N_MAX_Y-2;y++)
286          {
287              HHP_PRIME[x][y]=0.1*sqrt(x*x+y*y);
288          }
289      }
290
291      //This represents the part of HHP that is a function of "R" and "Z" (
292      NOT "PSI"). This includes any possible constant term.
293      __forceinline T f(int m, int n)
294      {
295          return 0;
296      }
297      #undef i
298      #undef j

```

F.2.2 ckf.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE STANDARD
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      /*
69      //Unused
70      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
71      //c[#] is the coefficient for (Pij)^(#+1)
72      c[0][s][t]=0;
73      c[1][s][t]=0;
74      c[2][s][t]=0;
75      */
76      }
77  }
78
79  //Left Boundary P=0
80  for (int t=1;t<=N_MAX_Y-4;t++)
81  {
82      int s=0;
83
84      bL1[t]=1;
85      bL2[t]=0;
86      bL3[t]=0;
87      bL4[t]=0;
88      bL5[t]=0;
89      bL6[t]=0;
90  }

```

```

91
92 //Right Boundary RPr+ZPz=0
93 for(int t=1;t<=N_MAX_Y-4;t++)
94 {
95     int s=N_MAX_X-3;
96
97     bR1[t]=-i;
98     bR2[t]=i;
99     bR3[t]=-j;
100    bR4[t]=j;
101    bR5[t]=0;
102    bR6[t]=0;
103 }
104
105 //Bottom Boundary
106 //Outside star, inside light cylinder Pz=0
107 for(int s=1; ( (s<N_LC-1) && (s<=N_MAX_X-4) );s++)
108 {
109     int t=0;
110
111     bB1[s]=0;
112     bB2[s]=0;
113     bB3[s]=1;
114     bB4[s]=0;
115     bB5[s]=-1;
116     bB6[s]=0;
117 }
118
119 //Outside light cylinder P=P_OP which is specified from the
    start.
120 for(int s=N_LC-1;s<=N_MAX_X-4;s++)
121 {
122     int t=0;
123
124     bB1[s]=0;
125     bB2[s]=0;
126     bB3[s]=1;
127     bB4[s]=0;
128     bB5[s]=0;
129     bB6[s]=P_OP;
130 }
131
132 //Top Boundary RPr+ZPz=0
133 for(int s=1;s<=N_MAX_X-4;s++)
134 {
135     int t=N_MAX_Y-3;
136
137     bT1[s]=-i;
138     bT2[s]=i;
139     bT3[s]=-j;
140     bT4[s]=j;
141     bT5[s]=0;

```

```

142             bT6[s]=0;
143     }
144
145     //Bottom Left Corner
146     {
147         int s=0;
148         int t=0;
149
150         bL1[t]=1;
151         bL2[t]=0;
152         bL3[t]=0;
153         bL4[t]=0;
154         bL5[t]=0;
155         bL6[t]=0;
156
157         bB1[s]=0;
158         bB2[s]=0;
159         bB3[s]=1;
160         bB4[s]=0;
161         bB5[s]=0;
162         bB6[s]=P_OP;
163     }
164
165     //Top Left Corner
166     {
167         int s=0;
168         int t=N_MAX_Y-3;
169
170         bL1[t]=1;
171         bL2[t]=0;
172         bL3[t]=0;
173         bL4[t]=0;
174         bL5[t]=0;
175         bL6[t]=0;
176
177         bT1[s]=0;
178         bT2[s]=i;
179         bT3[s]=-j;
180         bT4[s]=j;
181         bT5[s]=0;
182         bT6[s]=0;
183     }
184
185     //Bottom Right Corner
186     {
187         int s=N_MAX_X-3;
188         int t=0;
189
190         bR1[t]=-i;
191         bR2[t]=i;
192         bR3[t]=0;
193         bR4[t]=j;

```

```

194         bR5[t]=0;
195         bR6[t]=j*P_OP;
196
197         bB1[s]=0;
198         bB2[s]=0;
199         bB3[s]=1;
200         bB4[s]=0;
201         bB5[s]=0;
202         bB6[s]=P_OP;
203     }
204
205     //Top Right Corner
206     {
207         int s=N_MAX_X-3;
208         int t=N_MAX_Y-3;
209
210         bR1[t]=-1;
211         bR2[t]=1;
212         bR3[t]=0;
213         bR4[t]=0;
214         bR5[t]=0;
215         bR6[t]=0;
216
217         bT1[s]=0;
218         bT2[s]=0;
219         bT3[s]=-1;
220         bT4[s]=1;
221         bT5[s]=0;
222         bT6[s]=0;
223     }
224 }
225
226 //This function inserts the star.
227 void star(void)
228 {
229     for(int s=0;s<N_S_X;s++)
230     {
231         const T R=DX*i;
232         for(int t=0;t<N_S_Y;t++)
233         {
234             const T Z=DY*j;
235             a1[s][t]=0;
236             a2[s][t]=0;
237             a3[s][t]=0;
238             a4[s][t]=0;
239             a5[s][t]=1;
240             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
241         }
242     }
243 }
244

```

```

245 //This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
246 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
247 {
248     T HHP_L[N_MAX_Y-2];
249     for(int KP=0;KP<N_MAX_Y-2;KP++)
250     {
251         HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(
            N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
252     }
253
254     for(int jj=N_LC-1;jj==N_LC-1;jj++)
255     {
256         for(int kk=0;kk<=N_MAX_Y-3;kk++)
257         {
258             v0[RN(N_LC-1,kk)]=0.5*(v0[RN(N_LC-2,kk)]+v0[
                RN(N_LC,kk)]);
259         }
260     }
261
262     //at r<rL, there are many possible cases (closed field lines,
        open field lines that do or do not cross the light
        cylinder)
263     //for 0<r<=NR_S, we stay OUTSIDE star, because the star value
        is known and does not need to be altered.
264     for(int jj=0;jj<=N_S_X-1;jj++)
265     {
266         for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
267         {
268             int KP=0;
269             {
270                 const T PT=v0[RN(jj,kk)];
271
272                 if(PT<v0[RN(N_LC-2,0)])
273                 {
274                     if(PT<v0[RN(N_LC-2,N_MAX_Y-3)]
                        ])
275                     {
276                         HHP[jj][kk]=HHP_L[
                            N_MAX_Y-3]*PT/v0[
                                RN(N_LC-2,N_MAX_Y
                                    -3)];
277                     }
278                     else
279                     {
280                         for(;KP<N_MAX_Y-4;&&PT
                            <v0[RN(N_LC-2,KP)]
                                );KP++;
281
282                         const T Q1=PT-v0[RN(
                            N_LC-1,KP+1)];

```



```

283                                     const T Q2=PT-v0[RN(
284                                     N_LC-1,KP)];
                                     HHP[jj][kk]=(Q1*HHP_L
                                     [KP]-Q2*HHP_L[KP
                                     +1])/(Q1-Q2);
285                                     }
286                                     }
287                                     else
288                                     {
289                                     HHP[jj][kk]=0;
290                                     }
291                                     }
292                                     if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
293                                     }
294     }
295
296     //for NR_S<r<rLC
297     for(int jj=N_S_X;jj<=N_LC-2;jj++)
298     {
299         for(int kk=0;kk<=N_MAX_Y-3;kk++)
300         {
301             int KP=0;
302             {
303                 const T PT=v0[RN(jj,kk)];
304
305                 if(PT<v0[RN(N_LC-2,0)])
306                 {
307                     if(PT<v0[RN(N_LC-2,N_MAX_Y-3)
308                     ])
309                     {
310                         HHP[jj][kk]=HHP_L[
311                         N_MAX_Y-3]*PT/v0[
312                         RN(N_LC-2,N_MAX_Y
313                         -3)];
314                     }
315                     else
316                     {
317                         for(;KP<N_MAX_Y-4&&PT
318                         <v0[RN(N_LC-2,KP)
319                         ];KP++);
320
321                         const T Q1=PT-v0[RN(
322                         N_LC-1,KP+1)];
323                         const T Q2=PT-v0[RN(
324                         N_LC-1,KP)];
325                         HHP[jj][kk]=(Q1*HHP_L
326                         [KP]-Q2*HHP_L[KP
327                         +1])/(Q1-Q2);
328                     }
329                 }
330             }
331             else
332             {

```

```

322                                     HHP[jj][kk]=0;
323                                     }
324                                     }
325                                     if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
326                                 }
327                            }
328
329    for(int jj=N_LC-1;jj==N_LC-1;jj++)
330    {
331        for(int kk=0;kk<=N_MAX_Y-3;kk++)
332        {
333            HHP[N_LC-1][kk]=HHP_L[kk];
334        }
335    }
336
337    for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
338    {
339        for(int kk=0;kk<=N_MAX_Y-3;kk++)
340        {
341            int KP=0;
342            const T PT=v0[RN(jj,kk)];
343
344            for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++)
345                ;
346
347            const T Q1=PT-v0[RN(N_LC-1,KP+1)];
348            const T Q2=PT-v0[RN(N_LC-1,KP)];
349            //if(fabs(Q1-Q2)>0.00001)
350            {
351                HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP
352                    +1])/(Q1-Q2);
353            }
354            if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
355        }
356    }
357
358    //TODO - HHP_PRIME is largely unnecessary, and this
359    //calculation may be wrong.
360    for(int x=0;x<N_MAX_X-2;x++)
361    {
362        for(int y=0;y<N_MAX_Y-2;y++)
363        {
364            T part1,part2;
365            if(x==N_MAX_X-3)
366            {
367                part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(
368                    x,y)]-v0[RN(x-1,y)])/(DX*DX);
369            }
370            else
371            {
372                part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(
373                    x+1,y)]-v0[RN(x,y)])/(DX*DX);

```

```

369         }
370         if (y==N_MAX_Y-3)
371         {
372             part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(
                    x,y)]-v0[RN(x,y-1)])/(DY*DY);
373         }
374         else
375         {
376             part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(
                    x,y+1)]-v0[RN(x,y)])/(DY*DY);
377         }
378         HHP_PRIME[x][y]=part1+part2;
379     }
380 }
381 }
382
383 //This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
384 __forceinline T f(int m, int n)
385 {
386     return 0;
387 }
388
389 #undef i
390 #undef j

```

F.2.3 ckf_jets.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE STANDARD
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32
33             /*
34             //1st derivative is backward difference
35             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
36             a2[s][t]=1-i*i*DX*DX;
37             a3[s][t]=1-i*i*DX*DX;
38             a4[s][t]=1-i*i*DX*DX;
39             a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
40             a6[s][t]=0;
41             */
42
43             //1st derivative is backward difference, DX
44             and DY independent
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
47             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
48             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```

44         a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
45             -1/i-i*DX*DX;
46         a6[s][t]=0;
47         //Value near LC is average of values to the
48             left and right.
49         if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
50         {
51             a1[s][t]=-0.5;
52             a2[s][t]=-0.5;
53             a3[s][t]=0;
54             a4[s][t]=0;
55             a5[s][t]=1;
56             a6[s][t]=0;
57         }
58         /*
59         //Unused
60         //Polynomial coefficients of HHP (constant
61             term taken care of in function "f")
62         //c[#] is the coefficient for (Pij)^(#+1)
63         c[0][s][t]=0;
64         c[1][s][t]=0;
65         c[2][s][t]=0;
66         */
67     }
68
69     //Left Boundary P=0
70     for (int t=1;t<=N_MAX_Y-4;t++)
71     {
72         int s=0;
73
74         bL1[t]=1;
75         bL2[t]=0;
76         bL3[t]=0;
77         bL4[t]=0;
78         bL5[t]=0;
79         bL6[t]=0;
80     }
81
82     //Right Boundary
83     for (int t=1;t<=N_MAX_Y-4;t++)
84     {
85         int s=N_MAX_X-3;
86     #ifdef JETS1
87         //RPr+ZPz=0
88         bR1[t]=-i;
89         bR2[t]=i;
90         bR3[t]=-j;
91         bR4[t]=j;
92         bR5[t]=0;

```

```

93             bR6[t]=0;
94 #endif
95 #ifdef JETS2
96             //Prr=0
97             bR1[t]=1;
98             bR2[t]=1;
99             bR3[t]=0;
100            bR4[t]=0;
101            bR5[t]=-2;
102            bR6[t]=0;
103 #endif
104     }
105
106     //Bottom Boundary
107     //Outside star, inside light cylinder Pz=0
108     for(int s=1; ((s<N_LC-1) && (s<=N_MAX_X-4)); s++)
109     {
110         int t=0;
111
112         bB1[s]=0;
113         bB2[s]=0;
114         bB3[s]=1;
115         bB4[s]=0;
116         bB5[s]=-1;
117         bB6[s]=0;
118     }
119
120     //Outside light cylinder P=P_OP which is specified from the
121     start.
122     for(int s=N_LC-1; s<=N_MAX_X-4; s++)
123     {
124         int t=0;
125
126         bB1[s]=0;
127         bB2[s]=0;
128         bB3[s]=1;
129         bB4[s]=0;
130         bB5[s]=0;
131         bB6[s]=P_OP;
132     }
133     //Top Boundary
134     for(int s=1; s<=N_MAX_X-4; s++)
135     {
136         int t=N_MAX_Y-3;
137 #ifdef JETS1
138         // RPr+ZPz=0
139         bT1[s]=-i;
140         bT2[s]=i;
141         bT3[s]=-j;
142         bT4[s]=j;
143         bT5[s]=0;

```

```

144             bT6[s]=0;
145 #endif
146 #ifdef JETS2
147             //Prr=0
148             bT1[s]=0;
149             bT2[s]=0;
150             bT3[s]=1;
151             bT4[s]=1;
152             bT5[s]=-2;
153             bT6[s]=0;
154 #endif
155     }
156
157     //Bottom Left Corner
158     {
159         int s=0;
160         int t=0;
161
162         bL1[t]=1;
163         bL2[t]=0;
164         bL3[t]=0;
165         bL4[t]=0;
166         bL5[t]=0;
167         bL6[t]=0;
168
169         bB1[s]=0;
170         bB2[s]=0;
171         bB3[s]=1;
172         bB4[s]=0;
173         bB5[s]=0;
174         bB6[s]=P_OP;
175     }
176
177     //Top Left Corner
178     {
179         int s=0;
180         int t=N_MAX_Y-3;
181
182         bL1[t]=1;
183         bL2[t]=0;
184         bL3[t]=0;
185         bL4[t]=0;
186         bL5[t]=0;
187         bL6[t]=0;
188
189 #ifdef JETS1
190         bT1[s]=0;
191         bT2[s]=i;
192         bT3[s]=-j;
193         bT4[s]=j;
194         bT5[s]=0;
195         bT6[s]=0;

```

```

196 #endif
197 #ifdef JETS2
198     bT1[s]=0;
199     bT2[s]=0;
200     bT3[s]=1;
201     bT4[s]=1;
202     bT5[s]=-2;
203     bT6[s]=0;
204 #endif
205     }
206
207     //Bottom Right Corner
208     {
209         int s=N_MAX_X-3;
210         int t=0;
211
212     #ifdef JETS1
213         bR1[t]=-i;
214         bR2[t]=i;
215         bR3[t]=0;
216         bR4[t]=j;
217         bR5[t]=0;
218         bR6[t]=j*P_OP;
219     #endif
220     #ifdef JETS2
221         bR1[t]=1;
222         bR2[t]=1;
223         bR3[t]=0;
224         bR4[t]=0;
225         bR5[t]=-2;
226         bR6[t]=0;
227     #endif
228
229         bB1[s]=0;
230         bB2[s]=0;
231         bB3[s]=1;
232         bB4[s]=0;
233         bB5[s]=0;
234         bB6[s]=P_OP;
235     }
236
237     //Top Right Corner
238     {
239         int s=N_MAX_X-3;
240         int t=N_MAX_Y-3;
241
242     #ifdef JETS1
243         bR1[t]=-1;
244         bR2[t]=1;
245         bR3[t]=0;
246         bR4[t]=0;
247         bR5[t]=0;

```



```

248             bR6[t]=0;
249
250             bT1[s]=0;
251             bT2[s]=0;
252             bT3[s]=-1;
253             bT4[s]=1;
254             bT5[s]=0;
255             bT6[s]=0;
256 #endif
257 #ifdef JETS2
258             bR1[t]=1;
259             bR2[t]=1;
260             bR3[t]=0;
261             bR4[t]=0;
262             bR5[t]=-2;
263             bR6[t]=0;
264
265             bT1[s]=0;
266             bT2[s]=0;
267             bT3[s]=1;
268             bT4[s]=1;
269             bT5[s]=-2;
270             bT6[s]=0;
271 #endif
272     }
273 }
274
275 //This function inserts the star.
276 void star(void)
277 {
278     for(int s=0;s<N_S_X;s++)
279     {
280         const T R=DX*i;
281         for(int t=0;t<N_S_Y;t++)
282         {
283             const T Z=DY*j;
284             a1[s][t]=0;
285             a2[s][t]=0;
286             a3[s][t]=0;
287             a4[s][t]=0;
288             a5[s][t]=1;
289             //a6[s][t]=R*R/pow(R*R+Z*Z,1.5); //Typical
                choice for the star.
290             a6[s][t]=1.0/pow(R*R+Z*Z,0.5); //Lovelace
                used this choice instead for the jets case
                .
291         }
292     }
293 }
294
295 //This fills in HHP and d(HHP)/d(PHI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.

```

```

296 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
297 {
298     T KH=2.48;
299     //T BETA=0.9995;
300
301     T HHPCORN=(v0[RN(N_LC,N_MAX_Y-3)]-v0[RN(N_LC-2,N_MAX_Y-3)])
        *(1/DX);
302     //T BETA=0.5*(3-sqrt(1+(8*v0[RN(N_LC-2,N_MAX_Y-3)]*HHPCORN)/(
        KH*KH)));
303     //T BETA=2.0;
304     printf("BETA=%f\n",BETA);
305
306     T HHP_L[N_MAX_Y-2];
307     for(int KP=0;KP<N_MAX_Y-2;KP++)
308     {
309         HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(
            N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
310     }
311
312     //at r<rL, there are many possible cases (closed field lines,
        open field lines that do or do not cross the light
        cylinder)
313     //for 0<r<=NR_S, we stay OUTSIDE star, because the star value
        is known and does not need to be altered.
314     for(int jj=0;jj<=N_S_X-1;jj++)
315     {
316         for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
317         {
318             int KP=0;
319             {
320                 const T PT=v0[RN(jj,kk)];
321
322                 if(PT<v0[RN(N_LC-2,0)])
323                 {
324                     if(PT<v0[RN(N_LC-2,N_MAX_Y-3)]
                        ])
325                     {
326                         const T PS=PT/v0[RN(
                            N_LC-2,N_MAX_Y-3)
                            ];
327                         HHP[jj][kk]=KH*KH*(PS
                            -0.5*BETA*PS*PS)
                            *(1-BETA*PS);
328                     }
329                     else if(PT<v0[RN(N_LC-2,
                        N_MAX_Y-3)])
330                     {
331                         HHP[jj][kk]=HHP_L[
                            N_MAX_Y-3]*PT/v0[
                            RN(N_LC-2,N_MAX_Y
                                -3)];
332                     }

```

```

333                                     else
334                                     {
335                                     for (; KP<N_MAX_Y-4&&PT
                                     <v0 [RN (N_LC-2, KP)
                                     ]; KP++);
336
337                                     const T Q1=PT-v0 [RN (
                                     N_LC-1, KP+1) ];
338                                     const T Q2=PT-v0 [RN (
                                     N_LC-1, KP) ];
339                                     HHP [jj] [kk] = (Q1*HHP_L
                                     [KP] - Q2*HHP_L [KP
                                     +1]) / (Q1-Q2);
340                                     }
341                                     }
342                                     else
343                                     {
344                                     HHP [jj] [kk]=0;
345                                     }
346                                     }
347                                     if (isnan (HHP [jj] [kk])) {HHP [jj] [kk]=0;}
348                                     }
349                                     }
350
351                                     //for NR_S<r<rLC
352                                     for (int jj=N_S_X; jj<=N_LC-2; jj++)
353                                     {
354                                     for (int kk=0; kk<=N_MAX_Y-3; kk++)
355                                     {
356                                     int KP=0;
357                                     {
358                                     const T PT=v0 [RN (jj, kk) ];
359
360                                     if (PT<v0 [RN (N_LC-2, 0) ])
361                                     {
362                                     if (PT<v0 [RN (N_LC-2, N_MAX_Y-3)
363                                     ])
364                                     {
365                                     const T PS=PT/v0 [RN (
366                                     N_LC-2, N_MAX_Y-3)
367                                     ];
368                                     HHP [jj] [kk] = KH*KH* (PS
369                                     -0.5*BETA*PS*PS)
370                                     * (1-BETA*PS);
371                                     }
372                                     else if (PT<v0 [RN (N_LC-2,
373                                     N_MAX_Y-3) ])
374                                     {
375                                     HHP [jj] [kk] = HHP_L [
376                                     N_MAX_Y-3] *PT/v0 [
377                                     RN (N_LC-2, N_MAX_Y
378                                     -3) ];

```

```

370                                     }
371                                     else
372                                     {
373                                         for (; KP<N_MAX_Y-4&&PT
<v0 [RN (N_LC-2, KP)
                                         ]; KP++);
374
375                                         const T Q1=PT-v0 [RN (
N_LC-1, KP+1) ];
376                                         const T Q2=PT-v0 [RN (
N_LC-1, KP) ];
377                                         HHP [jj] [kk] = (Q1*HHP_L
[KP] -Q2*HHP_L [KP
+1]) / (Q1-Q2);
378                                     }
379                                     }
380                                     else
381                                     {
382                                         HHP [jj] [kk]=0;
383                                     }
384                                     }
385                                     if (isnan (HHP [jj] [kk])) {HHP [jj] [kk]=0;}
386                                 }
387                            }
388
389                            for (int jj=N_LC-1; jj==N_LC-1; jj++)
390                            {
391                                for (int kk=0; kk<=N_MAX_Y-3; kk++)
392                                {
393                                    HHP [N_LC-1] [kk]=HHP_L [kk];
394                                }
395                            }
396
397                            for (int jj=N_LC; jj<=N_MAX_X-3; jj++)
398                            {
399                                for (int kk=0; kk<=N_MAX_Y-3; kk++)
400                                {
401                                    int KP=0;
402                                    const T PT=v0 [RN (jj, kk) ];
403
404                                    for (; KP<N_MAX_Y-4&&PT<v0 [RN (N_LC-2, KP) ]; KP++)
405                                        ;
406
407                                    const T Q1=PT-v0 [RN (N_LC-1, KP+1) ];
408                                    const T Q2=PT-v0 [RN (N_LC-1, KP) ];
409                                    //if (fabs (Q1-Q2)>0.00001)
410                                    {
411                                        HHP [jj] [kk] = (Q1*HHP_L [KP] -Q2*HHP_L [KP
+1]) / (Q1-Q2);
412                                    }
413                                    if (isnan (HHP [jj] [kk])) {HHP [jj] [kk]=0;}

```

```

414     }
415
416     //TODO - HHP_PRIME is largely unnecessary, and this
417     calculation may be wrong.
418     for(int x=0;x<N_MAX_X-2;x++)
419     {
420         for(int y=0;y<N_MAX_Y-2;y++)
421         {
422             T part1,part2;
423             if(x==N_MAX_X-3)
424             {
425                 part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(
426                     x,y)]-v0[RN(x-1,y)])/(DX*DX);
427             }
428             else
429             {
430                 part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(
431                     x+1,y)]-v0[RN(x,y)])/(DX*DX);
432             }
433             if(y==N_MAX_Y-3)
434             {
435                 part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(
436                     x,y)]-v0[RN(x,y-1)])/(DY*DY);
437             }
438             else
439             {
440                 part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(
441                     x,y+1)]-v0[RN(x,y)])/(DY*DY);
442             }
443             HHP_PRIME[x][y]=part1+part2;
444         }
445     }
446 }
447
448 //This represents the part of HHP that is a function of "R" and "Z" (
449 NOT "PSI"). This includes any possible constant term.
450 __forceinline T f(int m, int n)
451 {
452     return 0;
453 }
454
455 #undef i
456 #undef j

```

F.2.4 ckf_null.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE NULLSHEET
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      /*
69      //Unused
70      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
71      //c[#] is the coefficient for (Pij)^(#+1)
72      c[0][s][t]=0;
73      c[1][s][t]=0;
74      c[2][s][t]=0;
75      */
76      }
77  }
78
79  //The bottom right coner and bottom edge past the light
      cylinder are changed elsewhere, so the choices for these
      are largely unimportant.
80  //Left Boundary P=0
81  for(int t=1;t<=N_MAX_Y-4;t++)
82  {
83      int s=0;
84
85      bL1[t]=1;
86      bL2[t]=0;
87      bL3[t]=0;
88      bL4[t]=0;

```

```

89             bL5[t]=0;
90             bL6[t]=0;
91         }
92
93         //Right Boundary RPr+ZPz=0
94         for(int t=1;t<=N_MAX_Y-4;t++)
95         {
96             int s=N_MAX_X-3;
97
98             bR1[t]=-i;
99             bR2[t]=i;
100            bR3[t]=-j;
101            bR4[t]=j;
102            bR5[t]=0;
103            bR6[t]=0;
104        }
105
106        //Bottom Boundary
107        //Outside star, inside light cylinder Pz=0
108        for(int s=1; ((s<=N_LC-1)&&(s<=N_MAX_X-4));s++)
109        {
110            int t=0;
111
112            bB1[s]=0;
113            bB2[s]=0;
114            bB3[s]=1;
115            bB4[s]=0;
116            bB5[s]=-1;
117            bB6[s]=0;
118        }
119
120        //Outside light cylinder  $H^2=(R^2-1)*(Pz)^2$ 
121        //To begin the simulation, just use  $P=P_{OP}$ 
122        for(int s=N_LC;s<=N_MAX_X-4;s++)
123        {
124            int t=0;
125
126            bB1[s]=0;
127            bB2[s]=0;
128            bB3[s]=1;
129            bB4[s]=0;
130            bB5[s]=0;
131            bB6[s]=P_OP;
132        }
133
134        //Top Boundary RPr+ZPz=0
135        for(int s=1;s<=N_MAX_X-4;s++)
136        {
137            int t=N_MAX_Y-3;
138
139            bT1[s]=-i;
140            bT2[s]=i;

```



```

141         bT3[s]=-j;
142         bT4[s]=j;
143         bT5[s]=0;
144         bT6[s]=0;
145     }
146
147     //Bottom Left Corner
148     {
149         int s=0;
150         int t=0;
151
152         bL1[t]=1;
153         bL2[t]=0;
154         bL3[t]=0;
155         bL4[t]=0;
156         bL5[t]=0;
157         bL6[t]=0;
158
159         bB1[s]=0;
160         bB2[s]=0;
161         bB3[s]=1;
162         bB4[s]=0;
163         bB5[s]=0;
164         bB6[s]=P_OP;
165     }
166
167     //Top Left Corner
168     {
169         int s=0;
170         int t=N_MAX_Y-3;
171
172         bL1[t]=1;
173         bL2[t]=0;
174         bL3[t]=0;
175         bL4[t]=0;
176         bL5[t]=0;
177         bL6[t]=0;
178
179         bT1[s]=0;
180         bT2[s]=i;
181         bT3[s]=-j;
182         bT4[s]=j;
183         bT5[s]=0;
184         bT6[s]=0;
185     }
186
187     //Bottom Right Corner
188     {
189         int s=N_MAX_X-3;
190         int t=0;
191
192         bR1[t]=-i;

```

```

193         bR2[t]=i;
194         bR3[t]=0;
195         bR4[t]=j;
196         bR5[t]=0;
197         bR6[t]=j*P_OP;
198
199         bB1[s]=0;
200         bB2[s]=0;
201         bB3[s]=1;
202         bB4[s]=0;
203         bB5[s]=0;
204         bB6[s]=P_OP;
205     }
206
207     //Top Right Corner
208     {
209         int s=N_MAX_X-3;
210         int t=N_MAX_Y-3;
211
212         bR1[t]=-1;
213         bR2[t]=1;
214         bR3[t]=0;
215         bR4[t]=0;
216         bR5[t]=0;
217         bR6[t]=0;
218
219         bT1[s]=0;
220         bT2[s]=0;
221         bT3[s]=-1;
222         bT4[s]=1;
223         bT5[s]=0;
224         bT6[s]=0;
225     }
226 }
227
228 //This function inserts the star.
229 void star(void)
230 {
231     for(int s=0;s<N_S_X;s++)
232     {
233         const T R=DX*i;
234         for(int t=0;t<N_S_Y;t++)
235         {
236             const T Z=DY*j;
237             a1[s][t]=0;
238             a2[s][t]=0;
239             a3[s][t]=0;
240             a4[s][t]=0;
241             a5[s][t]=1;
242             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
243         }
244     }

```

```

245 }
246
247 //This fills in HHP and d(HHP)/d(PHI), which is a function of "PHI"
    and may or may not include both linear and nonlinear terms.
248 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
249 {
250     T HHP_L[N_MAX_Y-2];
251     for(int KP=0;KP<N_MAX_Y-2;KP++)
252     {
253         HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(
            N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
254     }
255
256     //at r<rL, there are many possible cases (closed field lines,
        open field lines that do or do not cross the light
        cylinder)
257     //for 0<r<=NR_S, we stay OUTSIDE star, because the star value
        is known and does not need to be altered.
258     for(int jj=0;jj<=N_S_X-1;jj++)
259     {
260         for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
261         {
262             int KP=0;
263             {
264                 const T PT=v0[RN(jj,kk)];
265
266                 if(PT<v0[RN(N_LC-2,0)])
267                 {
268                     if(PT<v0[RN(N_LC-2,N_MAX_Y-3)
                        ])
269                     {
270                         HHP[jj][kk]=HHP_L[
                            N_MAX_Y-3]*PT/v0[
                                RN(N_LC-2,N_MAX_Y
                                    -3)];
271                     }
272                     else
273                     {
274                         for(;KP<N_MAX_Y-4&&PT
                            <v0[RN(N_LC-2,KP)
                                ];KP++);
275
276                         const T Q1=PT-v0[RN(
                            N_LC-1,KP+1)];
277                         const T Q2=PT-v0[RN(
                            N_LC-1,KP)];
278                         HHP[jj][kk]=(Q1*HHP_L
                            [KP]-Q2*HHP_L[KP
                                +1])/(Q1-Q2);
279                     }
280                 }
281             else

```

```

282                                     {
283                                     HHP[jj][kk]=0;
284                                     }
285                                 }
286                                if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
287                            }
288                        }
289
290                    //for NR_S<r<rLC
291                    for(int jj=N_S_X;jj<=N_LC-2;jj++)
292                    {
293                        for(int kk=0;kk<=N_MAX_Y-3;kk++)
294                        {
295                            int KP=0;
296                            {
297                                const T PT=v0[RN(jj,kk)];
298
299                                if(PT<v0[RN(N_LC-2,0)])
300                                {
301                                    if(PT<v0[RN(N_LC-2,N_MAX_Y-3)
302                                        ])
303                                        {
304                                            HHP[jj][kk]=HHP_L[
305                                                N_MAX_Y-3]*PT/v0[
306                                                    RN(N_LC-2,N_MAX_Y
307                                                        -3)];
308
309                                            }
310                                            else
311                                            {
312                                                for(;KP<N_MAX_Y-4&&PT
313                                                    <v0[RN(N_LC-2,KP)
314                                                        ];KP++);
315
316                                                const T Q1=PT-v0[RN(
317                                                    N_LC-1,KP+1)];
318                                                const T Q2=PT-v0[RN(
319                                                    N_LC-1,KP)];
320                                                HHP[jj][kk]=(Q1*HHP_L
321                                                    [KP]-Q2*HHP_L[KP
322                                                        +1])/(Q1-Q2);
323                                            }
324                                        }
325                                    }
326                                }
327                                else
328                                {
329                                    HHP[jj][kk]=0;
330                                }
331                            }
332                        }
333                    }
334                }
335            }
336
337            for(int jj=N_LC-1;jj==N_LC-1;jj++)

```

```

324     {
325         for(int kk=0;kk<=N_MAX_Y-3;kk++)
326         {
327             HHP[N_LC-1][kk]=HHP_L[kk];
328         }
329     }
330
331     for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
332     {
333         for(int kk=0;kk<=N_MAX_Y-3;kk++)
334         {
335             int KP=0;
336             const T PT=v0[RN(jj,kk)];
337
338             for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++)
339                 ;
340
341             const T Q1=PT-v0[RN(N_LC-1,KP+1)];
342             const T Q2=PT-v0[RN(N_LC-1,KP)];
343             //if(fabs(Q1-Q2)>0.00001)
344             {
345                 HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP
346                     +1])/(Q1-Q2);
347             }
348             if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
349         }
350     }
351     //TODO - HHP_PRIME is largely unnecessary, and this
352     //calculation may be wrong.
353     for(int x=0;x<N_MAX_X-2;x++)
354     {
355         for(int y=0;y<N_MAX_Y-2;y++)
356         {
357             T part1,part2;
358             if(x==N_MAX_X-3)
359             {
360                 part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(
361                     x,y)]-v0[RN(x-1,y)])/(DX*DX);
362             }
363             else
364             {
365                 part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(
366                     x+1,y)]-v0[RN(x,y)])/(DX*DX);
367             }
368             if(y==N_MAX_Y-3)
369             {
370                 part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(
371                     x,y)]-v0[RN(x,y-1)])/(DY*DY);
372             }
373             else
374             {

```

```

370                                     part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(
                                     x,y+1)]-v0[RN(x,y)])/(DY*DY);
371                                     }
372                                     HHP_PRIME[x][y]=part1+part2;
373                                     }
374                                     }
375
376                                     resetCKF();
377     }
378
379     //This represents the part of HHP that is a function of "R" and "Z" (
        NOT "PSI"). This includes any possible constant term.
380     __forceinline T f(int m, int n)
381     {
382         return 0;
383     }
384
385     #undef i
386     #undef j

```

F.2.5 tak_monopole_test.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE MONOPOLE
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32
33             /*
34             //1st derivative is backward difference
35             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
36             a2[s][t]=1-i*i*DX*DX;
37             a3[s][t]=1-i*i*DX*DX;
38             a4[s][t]=1-i*i*DX*DX;
39             a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
40             a6[s][t]=0;
41             */
42
43             //1st derivative is backward difference, DX
44             and DY independent
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
47             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
48             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```

44         a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
45             -1/i-i*DX*DX;
46         a6[s][t]=0;
47         //Value near LC is average of values to the
48             left and right.
49         if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
50         {
51             a1[s][t]=-0.5;
52             a2[s][t]=-0.5;
53             a3[s][t]=0;
54             a4[s][t]=0;
55             a5[s][t]=1;
56             a6[s][t]=0;
57         }
58         //Polynomial coefficients of HHP (constant
59             term taken care of in function "f")
60         //c[#] is the coefficient for (Pij)^(#+1)
61         c[0][s][t]=0;
62         c[1][s][t]=0;
63         c[2][s][t]=0;
64     }
65
66     //Left Boundary P=0
67     for (int t=1;t<=N_MAX_Y-4;t++)
68     {
69         int s=0;
70
71         bL1[t]=1;
72         bL2[t]=0;
73         bL3[t]=0;
74         bL4[t]=0;
75         bL5[t]=0;
76         bL6[t]=0;
77     }
78
79     //Right Boundary RPr+ZPz=0
80     for (int t=1;t<=N_MAX_Y-4;t++)
81     {
82         int s=N_MAX_X-3;
83
84         bR1[t]=-i;
85         bR2[t]=i;
86         bR3[t]=-j;
87         bR4[t]=j;
88         bR5[t]=0;
89         bR6[t]=0;
90     }
91
92     //Bottom Boundary P=P_OP

```



```

93     for (int s=1; s<=N_MAX_X-4; s++)
94     {
95         int t=0;
96
97         bB1[s]=0;
98         bB2[s]=0;
99         bB3[s]=1;
100        bB4[s]=0;
101        bB5[s]=0;
102        bB6[s]=P_OP;
103    }
104
105    //Top Boundary RPr+ZPz=0
106    for (int s=1; s<=N_MAX_X-4; s++)
107    {
108        int t=N_MAX_Y-3;
109
110        bT1[s]=-i;
111        bT2[s]=i;
112        bT3[s]=-j;
113        bT4[s]=j;
114        bT5[s]=0;
115        bT6[s]=0;
116    }
117
118    //Bottom Left Corner
119    {
120        int s=0;
121        int t=0;
122
123        bL1[t]=1;
124        bL2[t]=0;
125        bL3[t]=0;
126        bL4[t]=0;
127        bL5[t]=0;
128        bL6[t]=0;
129
130        bB1[s]=0;
131        bB2[s]=0;
132        bB3[s]=1;
133        bB4[s]=0;
134        bB5[s]=0;
135        bB6[s]=P_OP;
136    }
137
138    //Top Left Corner
139    {
140        int s=0;
141        int t=N_MAX_Y-3;
142
143        bL1[t]=1;
144        bL2[t]=0;

```

```

145         bL3[t]=0;
146         bL4[t]=0;
147         bL5[t]=0;
148         bL6[t]=0;
149
150         bT1[s]=0;
151         bT2[s]=i;
152         bT3[s]=-j;
153         bT4[s]=j;
154         bT5[s]=0;
155         bT6[s]=0;
156     }
157
158     //Bottom Right Corner
159     {
160         int s=N_MAX_X-3;
161         int t=0;
162
163         bR1[t]=-i;
164         bR2[t]=i;
165         bR3[t]=0;
166         bR4[t]=j;
167         bR5[t]=0;
168         bR6[t]=j*P_OP;
169
170         bB1[s]=0;
171         bB2[s]=0;
172         bB3[s]=1;
173         bB4[s]=0;
174         bB5[s]=0;
175         bB6[s]=P_OP;
176     }
177
178     //Top Right Corner
179     {
180         int s=N_MAX_X-3;
181         int t=N_MAX_Y-3;
182
183         bR1[t]=-1;
184         bR2[t]=1;
185         bR3[t]=0;
186         bR4[t]=0;
187         bR5[t]=0;
188         bR6[t]=0;
189
190         bT1[s]=0;
191         bT2[s]=0;
192         bT3[s]=-1;
193         bT4[s]=1;
194         bT5[s]=0;
195         bT6[s]=0;
196     }

```

```

197 }
198
199 //This function inserts the star.
200 void star(void) {}
201
202 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
203 {
204     for(int x=0;x<N_MAX_X-2;x++)
205     {
206         for(int y=0;y<N_MAX_Y-2;y++)
207         {
208             HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c[1][x][y]*
                v0[RN(x,y)]*v0[RN(x,y)]+c[2][x][y]*v0[RN(x
                ,y)]*v0[RN(x,y)]*v0[RN(x,y)];
209             HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN
                (x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)
                ];
210         }
211     }
212 }
213
214 //This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
215 __forceinline T f(int m, int n)
216 {
217     T R=m*DX;
218     T Z=n*DY;
219     T Q=P_OP*(1-Z/sqrt(Z*Z+R*R));
220     return 2*Q*(Q/P_OP-1)*(Q/P_OP-2);
221 }
222
223 #undef i
224 #undef j

```

F.2.6 tak_monopole.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE MONOPOLE
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is backward difference
34             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
35             a2[s][t]=1-i*i*DX*DX;
36             a3[s][t]=1-i*i*DX*DX;
37             a4[s][t]=1-i*i*DX*DX;
38             a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
39             a6[s][t]=0;
40             */
41             /*
42             //1st derivative is central difference, DX
43             and DY independent
44             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
45             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
46             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```

44      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
45      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
46      ;
47      a6[s][t]=0;
48      */
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
69      //c[#] is the coefficient for (Pij)^(#+1)
70      c[0][s][t]=4;
71      c[1][s][t]=-6;
72      c[2][s][t]=2;
73      }
74  }
75
76  //Left Boundary P=0
77  for(int t=1;t<=N_MAX_Y-4;t++)
78  {
79      int s=0;
80
81      bL1[t]=1;
82      bL2[t]=0;
83      bL3[t]=0;
84      bL4[t]=0;
85      bL5[t]=0;
86      bL6[t]=0;
87  }
88
89  //Right Boundary RPr+ZPz=0
90  for(int t=1;t<=N_MAX_Y-4;t++)

```

```

91     {
92         int s=N_MAX_X-3;
93
94         bR1[t]=-i;
95         bR2[t]=i;
96         bR3[t]=-j;
97         bR4[t]=j;
98         bR5[t]=0;
99         bR6[t]=0;
100     }
101
102     //Bottom Boundary P=P_OP
103     for (int s=1; s<=N_MAX_X-4; s++)
104     {
105         int t=0;
106
107         bB1[s]=0;
108         bB2[s]=0;
109         bB3[s]=1;
110         bB4[s]=0;
111         bB5[s]=0;
112         bB6[s]=P_OP;
113     }
114
115     //Top Boundary RPr+ZPz=0
116     for (int s=1; s<=N_MAX_X-4; s++)
117     {
118         int t=N_MAX_Y-3;
119
120         bT1[s]=-i;
121         bT2[s]=i;
122         bT3[s]=-j;
123         bT4[s]=j;
124         bT5[s]=0;
125         bT6[s]=0;
126     }
127
128     //Bottom Left Corner
129     {
130         int s=0;
131         int t=0;
132
133         bL1[t]=1;
134         bL2[t]=0;
135         bL3[t]=0;
136         bL4[t]=0;
137         bL5[t]=0;
138         bL6[t]=0;
139
140         bB1[s]=0;
141         bB2[s]=0;
142         bB3[s]=1;

```

```

143         bB4[s]=0;
144         bB5[s]=0;
145         bB6[s]=P_OP;
146     }
147
148     //Top Left Corner
149     {
150         int s=0;
151         int t=N_MAX_Y-3;
152
153         bL1[t]=1;
154         bL2[t]=0;
155         bL3[t]=0;
156         bL4[t]=0;
157         bL5[t]=0;
158         bL6[t]=0;
159
160         bT1[s]=0;
161         bT2[s]=i;
162         bT3[s]=-j;
163         bT4[s]=j;
164         bT5[s]=0;
165         bT6[s]=0;
166     }
167
168     //Bottom Right Corner
169     {
170         int s=N_MAX_X-3;
171         int t=0;
172
173         bR1[t]=-i;
174         bR2[t]=i;
175         bR3[t]=0;
176         bR4[t]=j;
177         bR5[t]=0;
178         bR6[t]=j*P_OP;
179
180         bB1[s]=0;
181         bB2[s]=0;
182         bB3[s]=1;
183         bB4[s]=0;
184         bB5[s]=0;
185         bB6[s]=P_OP;
186     }
187
188     //Top Right Corner
189     {
190         int s=N_MAX_X-3;
191         int t=N_MAX_Y-3;
192
193         bR1[t]=-1;
194         bR2[t]=1;

```

```

195             bR3[t]=0;
196             bR4[t]=0;
197             bR5[t]=0;
198             bR6[t]=0;
199
200             bT1[s]=0;
201             bT2[s]=0;
202             bT3[s]=-1;
203             bT4[s]=1;
204             bT5[s]=0;
205             bT6[s]=0;
206         }
207     }
208
209     //This function inserts the star.
210     void star(void){}
211
212     //__forceinline void hhpSet(T HHP[(N_MAX_X-2)][(N_MAX_Y-2)],T
213         HHP_PRIME[(N_MAX_X-2)][(N_MAX_Y-2)])
214     __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
215     {
216         for(int x=0;x<N_MAX_X-2;x++)
217         {
218             for(int y=0;y<N_MAX_Y-2;y++)
219             {
220                 HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c[1][x][y]*
221                     v0[RN(x,y)]*v0[RN(x,y)]+c[2][x][y]*v0[RN(x
222                     ,y)]*v0[RN(x,y)]*v0[RN(x,y)];
223                 HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][y]*v0[RN
224                     (x,y)]+3*c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)
225                     ];
226             }
227         }
228     }
229
230     //This represents the part of HHP that is a function of "R" and "Z" (
231         NOT "PSI"). This includes any possible constant term.
232     __forceinline T f(int m, int n)
233     {
234         return 0.0;
235     }
236
237     #undef i
238     #undef j

```


F.2.7 tak_monopole_jets.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE MONOPOLE
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
               finite difference choice for the first
               derivative.
19             //The upper right corner is impossible to
               solve for using this choice, so I don't
               recommend this option.
20             //1st derivative is central difference
21             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
22             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
23             a3[s][t]=1-i*i*DX*DX;
24             a4[s][t]=1-i*i*DX*DX;
25             a5[s][t]=-4+4*i*i*DX*DX;
26             a6[s][t]=0;
27             */
28
29             /*
30             //1st derivative is backward difference
31             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
32             a2[s][t]=1-i*i*DX*DX;
33             a3[s][t]=1-i*i*DX*DX;
34             a4[s][t]=1-i*i*DX*DX;
35             a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
36             a6[s][t]=0;
37             */
38
39             //1st derivative is backward difference, DX
               and DY independent
40             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
41             a2[s][t]=1-i*i*DX*DX;
42             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
43             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```

44         a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
45             -1/i-i*DX*DX;
46         a6[s][t]=0;
47         //Value near LC is average of values to the
48             left and right.
49         if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
50         {
51             a1[s][t]=-0.5;
52             a2[s][t]=-0.5;
53             a3[s][t]=0;
54             a4[s][t]=0;
55             a5[s][t]=1;
56             a6[s][t]=0;
57         }
58         //Polynomial coefficients of HHP (constant
59             term taken care of in function "f")
60         //c[#] is the coefficient for (Pij)^(#+1)
61         c[0][s][t]=4;
62         c[1][s][t]=-6;
63         c[2][s][t]=2;
64     }
65
66     //Left Boundary P=0
67     for (int t=1;t<=N_MAX_Y-4;t++)
68     {
69         int s=0;
70
71         bL1[t]=1;
72         bL2[t]=0;
73         bL3[t]=0;
74         bL4[t]=0;
75         bL5[t]=0;
76         bL6[t]=0;
77     }
78
79     //Right Boundary RPr+ZPz=0
80     for (int t=1;t<=N_MAX_Y-4;t++)
81     {
82         int s=N_MAX_X-3;
83
84         bR1[t]=-i;
85         bR2[t]=i;
86         bR3[t]=-j;
87         bR4[t]=j;
88         bR5[t]=0;
89         bR6[t]=0;
90     }
91
92     //Bottom Boundary P=P_OP

```

```

93     for (int s=1; s<=N_MAX_X-4; s++)
94     {
95         int t=0;
96
97         bB1[s]=0;
98         bB2[s]=0;
99         bB3[s]=1;
100        bB4[s]=0;
101        bB5[s]=0;
102        bB6[s]=P_OP;
103    }
104
105    //Top Boundary RPr+ZPz=0
106    for (int s=1; s<=N_MAX_X-4; s++)
107    {
108        int t=N_MAX_Y-3;
109
110        bT1[s]=-i;
111        bT2[s]=i;
112        bT3[s]=-j;
113        bT4[s]=j;
114        bT5[s]=0;
115        bT6[s]=0;
116    }
117
118    //Bottom Left Corner
119    {
120        int s=0;
121        int t=0;
122
123        bL1[t]=1;
124        bL2[t]=0;
125        bL3[t]=0;
126        bL4[t]=0;
127        bL5[t]=0;
128        bL6[t]=0;
129
130        bB1[s]=0;
131        bB2[s]=0;
132        bB3[s]=1;
133        bB4[s]=0;
134        bB5[s]=0;
135        bB6[s]=P_OP;
136    }
137
138    //Top Left Corner
139    {
140        int s=0;
141        int t=N_MAX_Y-3;
142
143        bL1[t]=1;
144        bL2[t]=0;

```

```

145             bL3[t]=0;
146             bL4[t]=0;
147             bL5[t]=0;
148             bL6[t]=0;
149
150             bT1[s]=0;
151             bT2[s]=i;
152             bT3[s]=-j;
153             bT4[s]=j;
154             bT5[s]=0;
155             bT6[s]=0;
156     }
157
158     //Bottom Right Corner
159     {
160         int s=N_MAX_X-3;
161         int t=0;
162
163         bR1[t]=-i;
164         bR2[t]=i;
165         bR3[t]=0;
166         bR4[t]=j;
167         bR5[t]=0;
168         bR6[t]=j*P_OP;
169
170         bB1[s]=0;
171         bB2[s]=0;
172         bB3[s]=1;
173         bB4[s]=0;
174         bB5[s]=0;
175         bB6[s]=P_OP;
176     }
177
178     //Top Right Corner
179     {
180         int s=N_MAX_X-3;
181         int t=N_MAX_Y-3;
182
183         bR1[t]=-1;
184         bR2[t]=1;
185         bR3[t]=0;
186         bR4[t]=0;
187         bR5[t]=0;
188         bR6[t]=0;
189
190         bT1[s]=0;
191         bT2[s]=0;
192         bT3[s]=-1;
193         bT4[s]=1;
194         bT5[s]=0;
195         bT6[s]=0;
196     }

```

```

197 }
198
199 //This function inserts the star.
200 void star(void){}
201
202 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
203 {
204     for(int x=0;x<N_MAX_X-2;x++)
205     {
206         for(int y=0;y<N_MAX_Y-2;y++)
207         {
208             if(v0[RN(x,y)]>=P_OP)
209             {
210                 HHP[x][y]=0;
211             }
212             else if(v0[RN(x,y)]<v0[RN(N_LC-2,N_MAX_Y-3)])
213             {
214                 T KH=2.48;
215                 //T BETA=0.9995;
216                 //T HHPCORN=(v0[RN(N_LC,N_MAX_Y-3)] -
217                     v0[RN(N_LC-2,N_MAX_Y-3)])*(1/DX);
218                 //T BETA=0.5*(3-sqrt(1+(8*v0[RN(N_LC
219                     -2,N_MAX_Y-3)]*HHPCORN)/(KH*KH)));
220                 HHP[x][y]=KH*KH*0.5*v0[RN(x,y)]*(BETA
221                     *v0[RN(x,y)]/P_OP-1)*(BETA*v0[RN(x
222                     ,y)]/P_OP-2);
223             }
224             else
225             {
226                 HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c
227                     [1][x][y]*v0[RN(x,y)]*v0[RN(x,y)]+
228                     c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)
229                     ]*v0[RN(x,y)];
230                 HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][
231                     y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(
232                     x,y)]*v0[RN(x,y)];
233             }
234         }
235     }
236 }
237
238 //This represents the part of HHP that is a function of "R" and "Z" (
239 NOT "PSI"). This includes any possible constant term.
240 __forceinline T f(int m, int n)
241 {
242     return 0.0;
243 }
244
245 #undef i
246 #undef j

```

F.2.8 tak.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE STANDARD
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43         a3[s][t]=1-i*i*DX*DX;
44         a4[s][t]=1-i*i*DX*DX;
45         a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46         a6[s][t]=0;
47         */
48
49         //1st derivative is backward difference, DX
           and DY independent
50         a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51         a2[s][t]=1-i*i*DX*DX;
52         a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53         a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54         a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
           -1/i-i*DX*DX;
55         a6[s][t]=0;
56
57         //Value near LC is average of values to the
           left and right.
58         if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59         {
60             a1[s][t]=-0.5;
61             a2[s][t]=-0.5;
62             a3[s][t]=0;
63             a4[s][t]=0;
64             a5[s][t]=1;
65             a6[s][t]=0;
66         }
67
68         //Polynomial coefficients of HHP (constant
           term taken care of in function "f")
69         //c[#] is the coefficient for (Pij)^(#+1)
70         const T A_A=1/(RATIO*P_OP*P_OP);
71         const T P_RET=RATIO*P_OP;
72
73         c[0][s][t]=A_A*P_OP*P_RET;
74         c[1][s][t]=-A_A*(P_OP+P_RET);
75         c[2][s][t]=A_A;
76     }
77 }
78
79 //Left Boundary P=0
80 for(int t=1;t<=N_MAX_Y-4;t++)
81 {
82     int s=0;
83
84     bL1[t]=1;
85     bL2[t]=0;
86     bL3[t]=0;
87     bL4[t]=0;
88     bL5[t]=0;
89     bL6[t]=0;
90 }

```

```

91
92 //Right Boundary RPr+ZPz=0
93 for(int t=1;t<=N_MAX_Y-4;t++)
94 {
95     int s=N_MAX_X-3;
96
97     bR1[t]=-i;
98     bR2[t]=i;
99     bR3[t]=-j;
100    bR4[t]=j;
101    bR5[t]=0;
102    bR6[t]=0;
103 }
104
105 //Bottom Boundary
106 //Outside star, inside light cylinder Pz=0
107 for(int s=1; ( (s<N_LC-1) && (s<=N_MAX_X-4) );s++)
108 {
109     int t=0;
110
111     bB1[s]=0;
112     bB2[s]=0;
113     bB3[s]=1;
114     bB4[s]=0;
115     bB5[s]=-1;
116     bB6[s]=0;
117 }
118
119 //Outside light cylinder P=P_OP which is specified from the
    start.
120 for(int s=N_LC-1;s<=N_MAX_X-4;s++)
121 {
122     int t=0;
123
124     bB1[s]=0;
125     bB2[s]=0;
126     bB3[s]=1;
127     bB4[s]=0;
128     bB5[s]=0;
129     bB6[s]=P_OP;
130 }
131
132 //Top Boundary RPr+ZPz=0
133 for(int s=1;s<=N_MAX_X-4;s++)
134 {
135     int t=N_MAX_Y-3;
136
137     bT1[s]=-i;
138     bT2[s]=i;
139     bT3[s]=-j;
140     bT4[s]=j;
141     bT5[s]=0;

```



```

142             bT6[s]=0;
143     }
144
145     //Bottom Left Corner
146     {
147         int s=0;
148         int t=0;
149
150         bL1[t]=1;
151         bL2[t]=0;
152         bL3[t]=0;
153         bL4[t]=0;
154         bL5[t]=0;
155         bL6[t]=0;
156
157         bB1[s]=0;
158         bB2[s]=0;
159         bB3[s]=1;
160         bB4[s]=0;
161         bB5[s]=0;
162         bB6[s]=P_OP;
163     }
164
165     //Top Left Corner
166     {
167         int s=0;
168         int t=N_MAX_Y-3;
169
170         bL1[t]=1;
171         bL2[t]=0;
172         bL3[t]=0;
173         bL4[t]=0;
174         bL5[t]=0;
175         bL6[t]=0;
176
177         bT1[s]=0;
178         bT2[s]=i;
179         bT3[s]=-j;
180         bT4[s]=j;
181         bT5[s]=0;
182         bT6[s]=0;
183     }
184
185     //Bottom Right Corner
186     {
187         int s=N_MAX_X-3;
188         int t=0;
189
190         bR1[t]=-i;
191         bR2[t]=i;
192         bR3[t]=0;
193         bR4[t]=j;

```

```

194         bR5[t]=0;
195         bR6[t]=j*P_OP;
196
197         bB1[s]=0;
198         bB2[s]=0;
199         bB3[s]=1;
200         bB4[s]=0;
201         bB5[s]=0;
202         bB6[s]=P_OP;
203     }
204
205     //Top Right Corner
206     {
207         int s=N_MAX_X-3;
208         int t=N_MAX_Y-3;
209
210         bR1[t]=-1;
211         bR2[t]=1;
212         bR3[t]=0;
213         bR4[t]=0;
214         bR5[t]=0;
215         bR6[t]=0;
216
217         bT1[s]=0;
218         bT2[s]=0;
219         bT3[s]=-1;
220         bT4[s]=1;
221         bT5[s]=0;
222         bT6[s]=0;
223     }
224 }
225
226 //This function inserts the star.
227 void star(void)
228 {
229     for(int s=0;s<N_S_X;s++)
230     {
231         const T R=DX*i;
232         for(int t=0;t<N_S_Y;t++)
233         {
234             const T Z=DY*j;
235             a1[s][t]=0;
236             a2[s][t]=0;
237             a3[s][t]=0;
238             a4[s][t]=0;
239             a5[s][t]=1;
240             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
241         }
242     }
243 }
244

```

```

245 //This fills in HHP and d(HHP)/d(PHI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
246 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
247 {
248     for(int x=0;x<N_MAX_X-2;x++)
249     {
250         for(int y=0;y<N_MAX_Y-2;y++)
251         {
252             if(v0[RN(x,y)]>=P_OP)
253             {
254                 HHP[x][y]=0;
255                 HHP_PRIME[x][y]=0;
256             }
257             else
258             {
259                 HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c
                    [1][x][y]*v0[RN(x,y)]*v0[RN(x,y)]+
                    c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)]
                    *v0[RN(x,y)];
260                 HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][
                    y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(
                    x,y)]*v0[RN(x,y)];
261             }
262         }
263     }
264 }
265
266 //This represents the part of HHP that is a function of "r" and "z" (
    NOT "Phi"). This includes any possible constant term.
267 __forceinline T f(int m, int n)
268 {
269     return 0;
270 }
271
272 #undef i
273 #undef j

```

F.2.9 tak_jets.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE STANDARD
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
               finite difference choice for the first
               derivative.
19             //The upper right corner is impossible to
               solve for using this choice, so I don't
               recommend this option.
20             //1st derivative is central difference
21             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
22             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
23             a3[s][t]=1-i*i*DX*DX;
24             a4[s][t]=1-i*i*DX*DX;
25             a5[s][t]=-4+4*i*i*DX*DX;
26             a6[s][t]=0;
27             */
28
29             /*
30             //1st derivative is central difference, DX
               and DY independent
31             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
32             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
33             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
34             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
35             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
               ;
36             a6[s][t]=0;
37             */
38
39             /*
40             //1st derivative is backward difference
41             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
42             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
69      //c[#] is the coefficient for (Pij)^(#+1)
70      const T A_A=1/(RATIO*P_OP*P_OP);
71      const T P_RET=RATIO*P_OP;
72
73      c[0][s][t]=A_A*P_OP*P_RET;
74      c[1][s][t]=-A_A*(P_OP+P_RET);
75      c[2][s][t]=A_A;
76      }
77  }
78
79  //Left Boundary P=0
80  for(int t=1;t<=N_MAX_Y-4;t++)
81  {
82      int s=0;
83
84      bL1[t]=1;
85      bL2[t]=0;
86      bL3[t]=0;
87      bL4[t]=0;
88      bL5[t]=0;
89      bL6[t]=0;
90  }

```

```

91
92 //Right Boundary RPr+ZPz=0
93 for(int t=1;t<=N_MAX_Y-4;t++)
94 {
95     int s=N_MAX_X-3;
96
97     bR1[t]=-i;
98     bR2[t]=i;
99     bR3[t]=-j;
100    bR4[t]=j;
101    bR5[t]=0;
102    bR6[t]=0;
103 }
104
105 //Bottom Boundary
106 //Outside star, inside light cylinder Pz=0
107 for(int s=1; ( (s<N_LC-1) && (s<=N_MAX_X-4) );s++)
108 {
109     int t=0;
110
111     bB1[s]=0;
112     bB2[s]=0;
113     bB3[s]=1;
114     bB4[s]=0;
115     bB5[s]=-1;
116     bB6[s]=0;
117 }
118
119 //Outside light cylinder P=P_OP which is specified from the
    start.
120 for(int s=N_LC-1;s<=N_MAX_X-4;s++)
121 {
122     int t=0;
123
124     bB1[s]=0;
125     bB2[s]=0;
126     bB3[s]=1;
127     bB4[s]=0;
128     bB5[s]=0;
129     bB6[s]=P_OP;
130 }
131
132 //Top Boundary RPr+ZPz=0
133 for(int s=1;s<=N_MAX_X-4;s++)
134 {
135     int t=N_MAX_Y-3;
136
137     bT1[s]=-i;
138     bT2[s]=i;
139     bT3[s]=-j;
140     bT4[s]=j;
141     bT5[s]=0;

```

```

142             bT6[s]=0;
143     }
144
145     //Bottom Left Corner
146     {
147         int s=0;
148         int t=0;
149
150         bL1[t]=1;
151         bL2[t]=0;
152         bL3[t]=0;
153         bL4[t]=0;
154         bL5[t]=0;
155         bL6[t]=0;
156
157         bB1[s]=0;
158         bB2[s]=0;
159         bB3[s]=1;
160         bB4[s]=0;
161         bB5[s]=0;
162         bB6[s]=P_OP;
163     }
164
165     //Top Left Corner
166     {
167         int s=0;
168         int t=N_MAX_Y-3;
169
170         bL1[t]=1;
171         bL2[t]=0;
172         bL3[t]=0;
173         bL4[t]=0;
174         bL5[t]=0;
175         bL6[t]=0;
176
177         bT1[s]=0;
178         bT2[s]=i;
179         bT3[s]=-j;
180         bT4[s]=j;
181         bT5[s]=0;
182         bT6[s]=0;
183     }
184
185     //Bottom Right Corner
186     {
187         int s=N_MAX_X-3;
188         int t=0;
189
190         bR1[t]=-i;
191         bR2[t]=i;
192         bR3[t]=0;
193         bR4[t]=j;

```

```

194         bR5[t]=0;
195         bR6[t]=j*P_OP;
196
197         bB1[s]=0;
198         bB2[s]=0;
199         bB3[s]=1;
200         bB4[s]=0;
201         bB5[s]=0;
202         bB6[s]=P_OP;
203     }
204
205     //Top Right Corner
206     {
207         int s=N_MAX_X-3;
208         int t=N_MAX_Y-3;
209
210         bR1[t]=-1;
211         bR2[t]=1;
212         bR3[t]=0;
213         bR4[t]=0;
214         bR5[t]=0;
215         bR6[t]=0;
216
217         bT1[s]=0;
218         bT2[s]=0;
219         bT3[s]=-1;
220         bT4[s]=1;
221         bT5[s]=0;
222         bT6[s]=0;
223     }
224 }
225
226 //This function inserts the star.
227 void star(void)
228 {
229     for(int s=0;s<N_S_X;s++)
230     {
231         const T R=DX*i;
232         for(int t=0;t<N_S_Y;t++)
233         {
234             const T Z=DY*j;
235             a1[s][t]=0;
236             a2[s][t]=0;
237             a3[s][t]=0;
238             a4[s][t]=0;
239             a5[s][t]=1;
240             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
241         }
242     }
243 }
244

```



```

245 //This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
246 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
247 {
248     for(int x=0;x<N_MAX_X-2;x++)
249     {
250         for(int y=0;y<N_MAX_Y-2;y++)
251         {
252             if(v0[RN(x,y)]>=P_OP)
253             {
254                 HHP[x][y]=0;
255                 HHP_PRIME[x][y]=0;
256             }
257             else if (v0[RN(x,y)]<v0[RN(N_LC-2,N_MAX_Y-3)] )
258             {
259
260                 T KH=2.0;
261                 //T BETA=0.9995;
262                 //T HHPCORN=(v0[RN(N_LC,N_MAX_Y-3)] -
263                     v0[RN(N_LC-2,N_MAX_Y-3)])*(1/DX);
264                 //T BETA=0.5*(3-sqrt(1+(8*v0[RN(N_LC
265                     -2,N_MAX_Y-3)]*HHPCORN)/(KH*KH)));
266
267                 HHP[x][y]=KH*KH*0.5*v0[RN(x,y)]*(BETA
268                     *v0[RN(x,y)]/P_OP-1)*(BETA*v0[RN(x
269                     ,y)]/P_OP-2);
270             }
271             else
272             {
273                 HHP[x][y]=c[0][x][y]*v0[RN(x,y)]+c
274                     [1][x][y]*v0[RN(x,y)]*v0[RN(x,y)]+
275                     c[2][x][y]*v0[RN(x,y)]*v0[RN(x,y)]
276                     *v0[RN(x,y)];
277                 HHP_PRIME[x][y]=c[0][x][y]+2*c[1][x][
278                     y]*v0[RN(x,y)]+3*c[2][x][y]*v0[RN(
279                     x,y)]*v0[RN(x,y)];
280             }
281         }
282     }
283 }
284
285 //This represents the part of HHP that is a function of "R" and "Z" (
286     NOT "PSI"). This includes any possible constant term.
287 __forceinline T f(int m, int n)
288 {
289     return 0;
290 }
291
292 #undef i
293 #undef j

```

F.2.10 tak_theory_jets.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE JETS
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             //These equations represent an alternate
18             //finite difference choice for the first
19             //derivative.
20
21             /*
22             //The upper right corner is impossible to
23             //solve for using this choice, so I don't
24             //recommend this option.
25
26             //1st derivative is central difference
27             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
28             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
29             a3[s][t]=1-i*i*DX*DX;
30             a4[s][t]=1-i*i*DX*DX;
31             a5[s][t]=-4+4*i*i*DX*DX;
32             a6[s][t]=0;
33             */
34
35             /*
36             //1st derivative is backward difference
37             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
38             a2[s][t]=1-i*i*DX*DX;
39             a3[s][t]=1-i*i*DX*DX;
40             a4[s][t]=1-i*i*DX*DX;
41             a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
42             a6[s][t]=0;
43             */
44
45             //1st derivative is backward difference, DX
46             //and DY independent
47             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
48             a2[s][t]=1-i*i*DX*DX;
49             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
50             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
```

```

44         a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
45             -1/i-i*DX*DX;
46         a6[s][t]=0;
47         //Value near LC is average of values to the
48             left and right.
49         if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
50         {
51             a1[s][t]=-0.5;
52             a2[s][t]=-0.5;
53             a3[s][t]=0;
54             a4[s][t]=0;
55             a5[s][t]=1;
56             a6[s][t]=0;
57         }
58         //Polynomial coefficients of HHP (constant
59             term taken care of in function "f")
60         //c[#] is the coefficient for (Pij)^(#+1)
61         c[0][s][t]=4;
62         c[1][s][t]=-6;
63         c[2][s][t]=2;
64     }
65
66     //Left Boundary P=0
67     for (int t=1;t<=N_MAX_Y-4;t++)
68     {
69         int s=0;
70
71         bL1[t]=1;
72         bL2[t]=0;
73         bL3[t]=0;
74         bL4[t]=0;
75         bL5[t]=0;
76         bL6[t]=0;
77     }
78
79     //Right Boundary P=P_OP
80     for (int t=1;t<=N_MAX_Y-4;t++)
81     {
82         int s=N_MAX_X-3;
83
84         bR1[t]=0;
85         bR2[t]=1;
86         bR3[t]=0;
87         bR4[t]=0;
88         bR5[t]=0;
89         bR6[t]=P_OP;
90     }
91
92     //Bottom Boundary P=P_OP

```

```

93     for (int s=1; s<=N_MAX_X-4; s++)
94     {
95         int t=0;
96
97         bB1[s]=0;
98         bB2[s]=0;
99         bB3[s]=1;
100        bB4[s]=0;
101        bB5[s]=0;
102        bB6[s]=P_OP;
103    }
104
105    //Top Boundary
106    //Inside light cylinder Pz=0
107    for (int s=1; ( (s<N_LC-1) && (s<=N_MAX_X-4) ); s++)
108    {
109        int t=N_MAX_Y-3;
110
111        bT1[s]=0;
112        bT2[s]=0;
113        bT3[s]=0;
114        bT4[s]=1;
115        bT5[s]=-1;
116        bT6[s]=0;
117    }
118
119    //Outside light cylinder P=P_OP which is specified from the
120    //start.
121    for (int s=N_LC-1; s<=N_MAX_X-4; s++)
122    {
123        int t=N_MAX_Y-3;
124
125        bT1[s]=0;
126        bT2[s]=0;
127        bT3[s]=0;
128        bT4[s]=1;
129        bT5[s]=0;
130        bT6[s]=P_OP;
131    }
132
133    //Bottom Left Corner
134    {
135        int s=0;
136        int t=0;
137
138        bL1[t]=1;
139        bL2[t]=0;
140        bL3[t]=0;
141        bL4[t]=0;
142        bL5[t]=0;
143        bL6[t]=0;

```

```

144         bB1[s]=0;
145         bB2[s]=0;
146         bB3[s]=1;
147         bB4[s]=0;
148         bB5[s]=0;
149         bB6[s]=P_OP;
150     }
151
152     //Top Left Corner
153     {
154         int s=0;
155         int t=N_MAX_Y-3;
156
157         bL1[t]=1;
158         bL2[t]=0;
159         bL3[t]=0;
160         bL4[t]=0;
161         bL5[t]=0;
162         bL6[t]=0;
163
164         bT1[s]=0;
165         bT2[s]=0;
166         bT3[s]=0;
167         bT4[s]=1;
168         bT5[s]=-1;
169         bT6[s]=0;
170     }
171
172     //Bottom Right Corner
173     {
174         int s=N_MAX_X-3;
175         int t=0;
176
177         bR1[t]=0;
178         bR2[t]=1;
179         bR3[t]=0;
180         bR4[t]=0;
181         bR5[t]=0;
182         bR6[t]=P_OP;
183
184         bB1[s]=0;
185         bB2[s]=0;
186         bB3[s]=1;
187         bB4[s]=0;
188         bB5[s]=0;
189         bB6[s]=P_OP;
190     }
191
192     //Top Right Corner
193     {
194         int s=N_MAX_X-3;
195         int t=N_MAX_Y-3;

```

```

196
197         bR1[t]=0;
198         bR2[t]=1;
199         bR3[t]=0;
200         bR4[t]=0;
201         bR5[t]=0;
202         bR6[t]=P_OP;
203
204         bT1[s]=0;
205         bT2[s]=0;
206         bT3[s]=0;
207         bT4[s]=1;
208         bT5[s]=0;
209         bT6[s]=P_OP;
210     }
211 }
212
213 //This function inserts the star.
214 void star(void) {}
215
216 //__forceinline void hhpSet (T HHP[ (N_MAX_X-2) ][ (N_MAX_Y-2) ], T
    HHP_PRIME[ (N_MAX_X-2) ][ (N_MAX_Y-2) ] )
217 __forceinline void hhpSet (T** HHP, T** HHP_PRIME)
218 {
219     for (int x=0; x<N_MAX_X-2; x++)
220     {
221         for (int y=0; y<N_MAX_Y-2; y++)
222         {
223             if (v0[RN(x,y)]>P_OP) {HHP[x][y]=0.0;}
224             else if (v0[RN(x,y)]<=0.0) {HHP[x][y]=0; v0[RN(
                x,y)]=0.0;}
225             else
226             {
227                 T KH=6.6;
228                 T BETA=1.6;
229                 HHP[x][y]=KH*KH*0.5*v0[RN(x,y)]*(BETA
                    *v0[RN(x,y)]/P_OP-1)*(BETA*v0[RN(x
                    ,y)]/P_OP-2);
230             }
231         }
232     }
233 }
234
235 //This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
236 __forceinline T f(int m, int n)
237 {
238     return 0.0;
239 }
240
241 #undef i
242 #undef j

```

F.2.11 tak_null.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE NULLSHEET
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      /*
69      //Unused
70      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
71      //c[#] is the coefficient for (Pij)^(#+1)
72      c[0][s][t]=0;
73      c[1][s][t]=0;
74      c[2][s][t]=0;
75      */
76      }
77  }
78
79  //The bottom right coner and bottom edge past the light
      cylinder are changed elsewhere, so the choices for these
      are largely unimportant.
80  //Left Boundary P=0
81  for(int t=1;t<=N_MAX_Y-4;t++)
82  {
83      int s=0;
84
85      bL1[t]=1;
86      bL2[t]=0;
87      bL3[t]=0;
88      bL4[t]=0;

```



```

89         bL5[t]=0;
90         bL6[t]=0;
91     }
92
93     //Right Boundary RPr+ZPz=0
94     for(int t=1;t<=N_MAX_Y-4;t++)
95     {
96         int s=N_MAX_X-3;
97
98         bR1[t]=-i;
99         bR2[t]=i;
100        bR3[t]=-j;
101        bR4[t]=j;
102        bR5[t]=0;
103        bR6[t]=0;
104    }
105
106    //Bottom Boundary
107    //Outside star, inside light cylinder Pz=0
108    for(int s=1; ((s<=N_LC-1)&&(s<=N_MAX_X-4));s++)
109    {
110        int t=0;
111
112        bB1[s]=0;
113        bB2[s]=0;
114        bB3[s]=1;
115        bB4[s]=0;
116        bB5[s]=-1;
117        bB6[s]=0;
118    }
119
120    //Outside light cylinder  $H^2=(R^2-1)*(Pz)^2$ 
121    //To begin the simulation, just use  $P=P_{OP}$ 
122    for(int s=N_LC;s<=N_MAX_X-4;s++)
123    {
124        int t=0;
125
126        bB1[s]=0;
127        bB2[s]=0;
128        bB3[s]=1;
129        bB4[s]=0;
130        bB5[s]=0;
131        bB6[s]=P_OP;
132    }
133
134    //Top Boundary RPr+ZPz=0
135    for(int s=1;s<=N_MAX_X-4;s++)
136    {
137        int t=N_MAX_Y-3;
138
139        bT1[s]=-i;
140        bT2[s]=i;

```

```

141         bT3[s]=-j;
142         bT4[s]=j;
143         bT5[s]=0;
144         bT6[s]=0;
145     }
146
147     //Bottom Left Corner
148     {
149         int s=0;
150         int t=0;
151
152         bL1[t]=1;
153         bL2[t]=0;
154         bL3[t]=0;
155         bL4[t]=0;
156         bL5[t]=0;
157         bL6[t]=0;
158
159         bB1[s]=0;
160         bB2[s]=0;
161         bB3[s]=1;
162         bB4[s]=0;
163         bB5[s]=0;
164         bB6[s]=P_OP;
165     }
166
167     //Top Left Corner
168     {
169         int s=0;
170         int t=N_MAX_Y-3;
171
172         bL1[t]=1;
173         bL2[t]=0;
174         bL3[t]=0;
175         bL4[t]=0;
176         bL5[t]=0;
177         bL6[t]=0;
178
179         bT1[s]=0;
180         bT2[s]=i;
181         bT3[s]=-j;
182         bT4[s]=j;
183         bT5[s]=0;
184         bT6[s]=0;
185     }
186
187     //Bottom Right Corner
188     {
189         int s=N_MAX_X-3;
190         int t=0;
191
192         bR1[t]=-i;

```

```

193         bR2[t]=i;
194         bR3[t]=0;
195         bR4[t]=j;
196         bR5[t]=0;
197         bR6[t]=j*P_OP;
198
199         bB1[s]=0;
200         bB2[s]=0;
201         bB3[s]=1;
202         bB4[s]=0;
203         bB5[s]=0;
204         bB6[s]=P_OP;
205     }
206
207     //Top Right Corner
208     {
209         int s=N_MAX_X-3;
210         int t=N_MAX_Y-3;
211
212         bR1[t]=-1;
213         bR2[t]=1;
214         bR3[t]=0;
215         bR4[t]=0;
216         bR5[t]=0;
217         bR6[t]=0;
218
219         bT1[s]=0;
220         bT2[s]=0;
221         bT3[s]=-1;
222         bT4[s]=1;
223         bT5[s]=0;
224         bT6[s]=0;
225     }
226 }
227
228 //This function inserts the star.
229 void star(void)
230 {
231     for(int s=0;s<N_S_X;s++)
232     {
233         const T R=DX*i;
234         for(int t=0;t<N_S_Y;t++)
235         {
236             const T Z=DY*j;
237             a1[s][t]=0;
238             a2[s][t]=0;
239             a3[s][t]=0;
240             a4[s][t]=0;
241             a5[s][t]=1;
242             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
243         }
244     }

```

```

245 }
246
247 //This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
248 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
249 {
250     for(int x=0;x<N_MAX_X-2;x++)
251     {
252         for(int y=0;y<N_MAX_Y-2;y++)
253         {
254             if(v0[RN(x,y)]>=P_OP){HHP[x][y]=0;}
255             else
256             {
257                 //All choices should work, but the
                    last one is the most general.
258                 //Original P_OP=1
259                 //HHP[x][y]=(1.07*v0[RN(x,y)]*(2-v0[
                    RN(x,y)]*pow(1-v0[RN(x,y)],0.4))
                    *(0.428*(5+6*v0[RN(x,y)]*(v0[RN(x,
                    y)]-2))/(pow(fabs(1-v0[RN(x,y)]
                    ,0.6))));
260                 //Alternate P_OP=1
261                 //HHP[x][y]=(1.07*v0[RN(x,y)]*(2-v0[
                    RN(x,y)]*pow(1-v0[RN(x,y)],0.4))
                    *(2.568*(v0[RN(x,y)]-1.40825)*(v0[
                    RN(x,y)]-0.591752))/(pow(fabs(1-v0
                    [RN(x,y)],0.6))));
262                 //General P_OP
263                 HHP[x][y]=(1.07*v0[RN(x,y)]*(2-v0[RN(
                    x,y)]/P_OP)*pow(1-v0[RN(x,y)]/P_OP
                    ,0.4))*(0.428*(5*P_OP*P_OP-12*P_OP
                    *v0[RN(x,y)]+6*v0[RN(x,y)]*v0[RN(x
                    ,y)]/(P_OP*P_OP*pow(fabs(1-v0[RN(
                    x,y)]/P_OP),0.6))));
264             }
265             HHP_PRIME[x][y]=0;
266         }
267     }
268     resetTak();
269 }
270
271 //This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
272 __forceinline T f(int m, int n)
273 {
274     return 0;
275 }
276
277 #undef i
278 #undef j

```

F.2.12 ckf_monopole_jets.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE MONOPOLE
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
69      //c[#] is the coefficient for (Pij)^(#+1)
70      c[0][s][t]=4;
71      c[1][s][t]=-6;
72      c[2][s][t]=2;
73      }
74  }
75
76  //Left Boundary P=0
77  for (int t=1;t<=N_MAX_Y-4;t++)
78  {
79      int s=0;
80
81      bL1[t]=1;
82      bL2[t]=0;
83      bL3[t]=0;
84      bL4[t]=0;
85      bL5[t]=0;
86      bL6[t]=0;
87  }
88
89  //Right Boundary RPr+ZPz=0
90  for (int t=1;t<=N_MAX_Y-4;t++)

```

```

91     {
92         int s=N_MAX_X-3;
93
94         bR1[t]=-i;
95         bR2[t]=i;
96         bR3[t]=-j;
97         bR4[t]=j;
98         bR5[t]=0;
99         bR6[t]=0;
100     }
101
102     //Bottom Boundary P=P_OP
103     for (int s=1; s<=N_MAX_X-4; s++)
104     {
105         int t=0;
106
107         bB1[s]=0;
108         bB2[s]=0;
109         bB3[s]=1;
110         bB4[s]=0;
111         bB5[s]=0;
112         bB6[s]=P_OP;
113     }
114
115     //Top Boundary RPr+ZPz=0
116     for (int s=1; s<=N_MAX_X-4; s++)
117     {
118         int t=N_MAX_Y-3;
119
120         bT1[s]=-i;
121         bT2[s]=i;
122         bT3[s]=-j;
123         bT4[s]=j;
124         bT5[s]=0;
125         bT6[s]=0;
126     }
127
128     //Bottom Left Corner
129     {
130         int s=0;
131         int t=0;
132
133         bL1[t]=1;
134         bL2[t]=0;
135         bL3[t]=0;
136         bL4[t]=0;
137         bL5[t]=0;
138         bL6[t]=0;
139
140         bB1[s]=0;
141         bB2[s]=0;
142         bB3[s]=1;

```

```

143         bB4[s]=0;
144         bB5[s]=0;
145         bB6[s]=P_OP;
146     }
147
148     //Top Left Corner
149     {
150         int s=0;
151         int t=N_MAX_Y-3;
152
153         bL1[t]=1;
154         bL2[t]=0;
155         bL3[t]=0;
156         bL4[t]=0;
157         bL5[t]=0;
158         bL6[t]=0;
159
160         bT1[s]=0;
161         bT2[s]=i;
162         bT3[s]=-j;
163         bT4[s]=j;
164         bT5[s]=0;
165         bT6[s]=0;
166     }
167
168     //Bottom Right Corner
169     {
170         int s=N_MAX_X-3;
171         int t=0;
172
173         bR1[t]=-i;
174         bR2[t]=i;
175         bR3[t]=0;
176         bR4[t]=j;
177         bR5[t]=0;
178         bR6[t]=j*P_OP;
179
180         bB1[s]=0;
181         bB2[s]=0;
182         bB3[s]=1;
183         bB4[s]=0;
184         bB5[s]=0;
185         bB6[s]=P_OP;
186     }
187
188     //Top Right Corner
189     {
190         int s=N_MAX_X-3;
191         int t=N_MAX_Y-3;
192
193         bR1[t]=-1;
194         bR2[t]=1;

```



```

195             bR3[t]=0;
196             bR4[t]=0;
197             bR5[t]=0;
198             bR6[t]=0;
199
200             bT1[s]=0;
201             bT2[s]=0;
202             bT3[s]=-1;
203             bT4[s]=1;
204             bT5[s]=0;
205             bT6[s]=0;
206         }
207     }
208
209     //This function inserts the star.
210     void star(void){}
211
212     //This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
213     and may or may not include both linear and nonlinear terms.
214     __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
215     {
216         T KH=8.48;
217         if (toggle)
218         {
219             printf("Takamori\n");
220
221             for(int x=0;x<N_MAX_X-2;x++)
222             {
223                 for(int y=0;y<N_MAX_Y-2;y++)
224                 {
225                     if (v0[RN(x,y)]<v0[RN(N_LC-2,N_MAX_Y
226                         -3)])
227                     {
228                         //T KH=2.48;
229                         //T BETA=0.9995;
230                         T HHPCORN=(v0[RN(N_LC,N_MAX_Y
231                             -3)]-v0[RN(N_LC-2,N_MAX_Y
232                             -3)])*(1/DX);
233                         T BETA=0.5*(3-sqrt(1+(8*v0[RN
234                             (N_LC-2,N_MAX_Y-3)]*
235                             HHPCORN)/(KH*KH))));
236                         HHP[x][y]=KH*KH*0.5*v0[RN(x,y)
237                             ]*(v0[RN(x,y)]-1)*(v0[RN(
238                             x,y)]-2);
239                         HHP_PRIME[x][y]=c[0][x][y]+2*
240                             c[1][x][y]*v0[RN(x,y)]+3*c
241                             [2][x][y]*v0[RN(x,y)]*v0[
242                             RN(x,y)];
243                     }
244                 }
245             }
246         }
247     }

```

```

235             HHP[x][y]=c[0][x][y]*v0[RN(x,
                y)]+c[1][x][y]*v0[RN(x,y)
                ]*v0[RN(x,y)]+c[2][x][y]*
                v0[RN(x,y)]*v0[RN(x,y)]*v0
                [RN(x,y)];
236             HHP_PRIME[x][y]=c[0][x][y]+2*
                c[1][x][y]*v0[RN(x,y)]+3*c
                [2][x][y]*v0[RN(x,y)]*v0[
                RN(x,y)];

237         }
238     }
239 }
240 }
241 else
242 {
243     printf("CKF\n");
244     //T KH=2.48;
245     //T BETA=0.9995;
246
247     T HHPCORN=(v0[RN(N_LC,N_MAX_Y-3)]-v0[RN(N_LC-2,
        N_MAX_Y-3)])*(1/DX);
248     T BETA=0.5*(3-sqrt(1+(8*v0[RN(N_LC-2,N_MAX_Y-3)]*
        HHPCORN)/(KH*KH)));
249
250     T HHP_L[N_MAX_Y-2];
251     for(int KP=0;KP<N_MAX_Y-2;KP++)
252     {
253         HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)
            ]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX
            );
254     }
255
256     //At r<rL, there are many possible cases (closed
        field lines, open field lines that do or do not
        cross the light cylinder).
257     //For 0<r<=NR_S, we stay OUTSIDE star, because the
        star value is known and does not need to be
        altered.
258
259     for(int jj=0;jj<=N_LC-2;jj++)
260     {
261         for(int kk=0;kk<=N_MAX_Y-3;kk++)
262         {
263             int KP=0;
264             {
265                 const T PT=v0[RN(jj,kk)];
266
267                 if(PT<v0[RN(N_LC-2,0)])
268                 {
269                     if(PT<v0[RN(N_LC-2,
                        N_MAX_Y-3)])
270                     {

```

```

271                                     const T PS=PT
                                         /v0[RN(
                                         N_LC-2,
                                         N_MAX_Y-3)
                                         ];
272                                     HHP[jj][kk]=
                                         KH*KH*(PS
                                         -0.5*BETA*
                                         PS*PS)*(1-
                                         BETA*PS);

273                                     }
274                                     else
275                                     {
276                                         for(;KP<
                                         N_MAX_Y
                                         -4&&PT<v0[
                                         RN(N_LC-2,
                                         KP)] ;KP++)
                                         ;

277                                     const T Q1=PT
                                         -v0[RN(
                                         N_LC-1,KP
                                         +1)];
278                                     const T Q2=PT
                                         -v0[RN(
                                         N_LC-1,KP)
                                         ];
279                                     HHP[jj][kk]=(
                                         Q1*HHP_L[
                                         KP]-Q2*
                                         HHP_L[KP
                                         +1])/ (Q1-
                                         Q2);

280                                     }
281                                     }
282                                     }
283                                     else
284                                     {
285                                         HHP[jj][kk]=0;
286                                     }
287                                     }
288                                     if(isnan(HHP[jj][kk])){HHP[jj][kk]
                                         ]=0;}

289                                     }
290                                     }
291                                     }
292                                     for(int jj=N_LC-1;jj==N_LC-1;jj++)
293                                     {
294                                         for(int kk=0;kk<=N_MAX_Y-3;kk++)
295                                         {
296                                             HHP[N_LC-1][kk]=HHP_L[kk];
297                                         }

```

```

298     }
299
300     for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
301     {
302         for(int kk=0;kk<=N_MAX_Y-3;kk++)
303         {
304             int KP=0;
305             const T PT=v0[RN(jj,kk)];
306
307             for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP
308                 )];KP++);
309
310             const T Q1=PT-v0[RN(N_LC-1,KP+1)];
311             const T Q2=PT-v0[RN(N_LC-1,KP)];
312             HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP
313                 +1])/(Q1-Q2);
314             if(isnan(HHP[jj][kk])){HHP[jj][kk
315                 ]=0;}
316         }
317     }
318
319     //TODO - HHP_PRIME is largely unnecessary, and this
320     //calculation may be wrong.
321     for(int x=0;x<N_MAX_X-2;x++)
322     {
323         for(int y=0;y<N_MAX_Y-2;y++)
324         {
325             HHP_PRIME[x][y]=0.1*sqrt(x*x+y*y);
326         }
327     }
328
329     //This represents the part of HHP that is a function of "R" and "Z" (
330     //NOT "PSI"). This includes any possible constant term.
331     __forceinline T f(int m, int n)
332     {
333         return 0;
334     }
335
336     #undef i
337     #undef j

```

F.2.13 ckf_tak.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE STANDARD
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if(s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
69      //c[#] is the coefficient for (Pij)^(#+1)
70      c[0][s][t]=0;
71      c[1][s][t]=0;
72      c[2][s][t]=0;
73      }
74  }
75
76  //Left Boundary P=0
77  for(int t=1;t<=N_MAX_Y-4;t++)
78  {
79      int s=0;
80
81      bL1[t]=1;
82      bL2[t]=0;
83      bL3[t]=0;
84      bL4[t]=0;
85      bL5[t]=0;
86      bL6[t]=0;
87  }
88
89  //Right Boundary RPr+ZPz=0
90  for(int t=1;t<=N_MAX_Y-4;t++)

```

```

91     {
92         int s=N_MAX_X-3;
93
94         bR1[t]=-i;
95         bR2[t]=i;
96         bR3[t]=-j;
97         bR4[t]=j;
98         bR5[t]=0;
99         bR6[t]=0;
100     }
101
102     //Bottom Boundary
103     //Outside star, inside light cylinder Pz=0
104     for(int s=1; ( (s<N_LC-1) && (s<=N_MAX_X-4) );s++)
105     {
106         int t=0;
107
108         bB1[s]=0;
109         bB2[s]=0;
110         bB3[s]=1;
111         bB4[s]=0;
112         bB5[s]=-1;
113         bB6[s]=0;
114     }
115
116     //Outside light cylinder P=P_OP which is specified from the
117     start.
118     for(int s=N_LC-1;s<=N_MAX_X-4;s++)
119     {
120         int t=0;
121
122         bB1[s]=0;
123         bB2[s]=0;
124         bB3[s]=1;
125         bB4[s]=0;
126         bB5[s]=0;
127         bB6[s]=P_OP;//Needs to be fixed
128     }
129
130     //Top Boundary RPr+ZPz=0
131     for(int s=1;s<=N_MAX_X-4;s++)
132     {
133         int t=N_MAX_Y-3;
134
135         bT1[s]=-i;
136         bT2[s]=i;
137         bT3[s]=-j;
138         bT4[s]=j;
139         bT5[s]=0;
140         bT6[s]=0;
141     }

```

```

142      //Bottom Left Corner
143      {
144          int s=0;
145          int t=0;
146
147          bL1[t]=1;
148          bL2[t]=0;
149          bL3[t]=0;
150          bL4[t]=0;
151          bL5[t]=0;
152          bL6[t]=0;
153
154          bB1[s]=0;
155          bB2[s]=0;
156          bB3[s]=1;
157          bB4[s]=0;
158          bB5[s]=0;
159          bB6[s]=P_OP;
160      }
161
162      //Top Left Corner
163      {
164          int s=0;
165          int t=N_MAX_Y-3;
166
167          bL1[t]=1;
168          bL2[t]=0;
169          bL3[t]=0;
170          bL4[t]=0;
171          bL5[t]=0;
172          bL6[t]=0;
173
174          bT1[s]=0;
175          bT2[s]=i;
176          bT3[s]=-j;
177          bT4[s]=j;
178          bT5[s]=0;
179          bT6[s]=0;
180      }
181
182      //Bottom Right Corner
183      {
184          int s=N_MAX_X-3;
185          int t=0;
186
187          bR1[t]=-i;
188          bR2[t]=i;
189          bR3[t]=0;
190          bR4[t]=j;
191          bR5[t]=0;
192          bR6[t]=j*P_OP;
193

```



```

194             bB1[s]=0;
195             bB2[s]=0;
196             bB3[s]=1;
197             bB4[s]=0;
198             bB5[s]=0;
199             bB6[s]=P_OP;
200     }
201
202     //Top Right Corner
203     {
204         int s=N_MAX_X-3;
205         int t=N_MAX_Y-3;
206
207         bR1[t]=-1;
208         bR2[t]=1;
209         bR3[t]=0;
210         bR4[t]=0;
211         bR5[t]=0;
212         bR6[t]=0;
213
214         bT1[s]=0;
215         bT2[s]=0;
216         bT3[s]=-1;
217         bT4[s]=1;
218         bT5[s]=0;
219         bT6[s]=0;
220     }
221 }
222
223 //This function inserts the star.
224 void star(void)
225 {
226     for(int s=0;s<N_S_X;s++)
227     {
228         const T R=DX*i;
229         for(int t=0;t<N_S_Y;t++)
230         {
231             const T Z=DY*j;
232             a1[s][t]=0;
233             a2[s][t]=0;
234             a3[s][t]=0;
235             a4[s][t]=0;
236             a5[s][t]=1;
237             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
238         }
239     }
240 }
241
242 //This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
243 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
244 {

```

```

245     if(toggle)
246     {
247         printf("Takamori\n");
248         //SMOOTH=-1; //Opportunity to change smoothness mid
                simulation. Change code so that SMOOTH is a
                variable, and not a macro.
249         return;
250     }
251     printf("CKF\n");
252
253     T HHP_L[N_MAX_Y-2];
254     for(int KP=0; KP<N_MAX_Y-2; KP++)
255     {
256         HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(
                N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
257     }
258
259     //at r<rL, there are many possible cases (closed field lines,
                open field lines that do or do not cross the light
                cylinder)
260     //for 0<r<=NR_S, we stay OUTSIDE star, because the star value
                is known and does not need to be altered.
261     for(int jj=0; jj<=N_S_X-1; jj++)
262     {
263         for(int kk=N_S_Y; kk<=N_MAX_Y-3; kk++)
264         {
265             int KP=0;
266             {
267                 const T PT=v0[RN(jj,kk)];
268
269                 if(PT<v0[RN(N_LC-2,0)])
270                 {
271                     if(PT<v0[RN(N_LC-2,N_MAX_Y-3)
                            ])
272                     {
273                         HHP[jj][kk]=HHP_L[
                            N_MAX_Y-3]*PT/v0[
                            RN(N_LC-2,N_MAX_Y
                                -3)];
274                     }
275                     else
276                     {
277                         for(; KP<N_MAX_Y-4&&PT
                            <v0[RN(N_LC-2,KP)
                                ]; KP++);
278
279                         const T Q1=PT-v0[RN(
                            N_LC-1,KP+1)];
280                         const T Q2=PT-v0[RN(
                            N_LC-1,KP)];
281                         HHP[jj][kk]=(Q1*HHP_L
                            [KP]-Q2*HHP_L[KP

```

```

282                                     +1)) / (Q1-Q2);
283                                     }
284                                     }
285                                     else
286                                     {
287                                     HHP[jj][kk]=0;
288                                     }
289                                     }
290                                     if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
291                                     }
292
293                                     //for NR_S<r<rLC
294                                     for(int jj=N_S_X;jj<=N_LC-2;jj++)
295                                     {
296                                     for(int kk=0;kk<=N_MAX_Y-3;kk++)
297                                     {
298                                     int KP=0;
299                                     {
300                                     const T PT=v0[RN(jj,kk)];
301
302                                     if(PT<v0[RN(N_LC-2,0)])
303                                     {
304                                     if(PT<v0[RN(N_LC-2,N_MAX_Y-3)
305                                     ])
306                                     {
307                                     HHP[jj][kk]=HHP_L[
308                                     N_MAX_Y-3]*PT/v0[
309                                     RN(N_LC-2,N_MAX_Y
310                                     -3)];
311                                     }
312                                     else
313                                     {
314                                     for(;KP<N_MAX_Y-4&&PT
315                                     <v0[RN(N_LC-2,KP)
316                                     ];KP++);
317
318                                     const T Q1=PT-v0[RN(
319                                     N_LC-1,KP+1)];
320                                     const T Q2=PT-v0[RN(
321                                     N_LC-1,KP)];
322                                     HHP[jj][kk]=(Q1*HHP_L
323                                     [KP]-Q2*HHP_L[KP
324                                     +1))/(Q1-Q2);
325                                     }
326                                     }
327                                     }
328                                     }
329                                     }
330                                     }
331                                     }
332                                     if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}

```

```

323     }
324 }
325
326 for(int jj=N_LC-1;jj==N_LC-1;jj++)
327 {
328     for(int kk=0;kk<=N_MAX_Y-3;kk++)
329     {
330         HHP[N_LC-1][kk]=HHP_L[kk];
331     }
332 }
333
334 for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
335 {
336     for(int kk=0;kk<=N_MAX_Y-3;kk++)
337     {
338         int KP=0;
339         const T PT=v0[RN(jj,kk)];
340
341         for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++)
342             ;
343
344         const T Q1=PT-v0[RN(N_LC-1,KP+1)];
345         const T Q2=PT-v0[RN(N_LC-1,KP)];
346         //if(fabs(Q1-Q2)>0.00001)
347         {
348             HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP
349                 +1])/(Q1-Q2);
350         }
351         if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
352     }
353 }
354
355 //TODO - HHP_PRIME is largely unnecessary, and this
356 //calculation may be wrong.
357 for(int x=0;x<N_MAX_X-2;x++)
358 {
359     for(int y=0;y<N_MAX_Y-2;y++)
360     {
361         T part1,part2;
362         if(x==N_MAX_X-3)
363         {
364             part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(
365                 x,y)]-v0[RN(x-1,y)])/(DX*DX);
366         }
367         else
368         {
369             part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(
370                 x+1,y)]-v0[RN(x,y)])/(DX*DX);
371         }
372         if(y==N_MAX_Y-3)
373         {

```

```

369                                     part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(
                                     x,y)]-v0[RN(x,y-1)])/(DY*DY);
370                                     }
371                                     else
372                                     {
373                                     part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(
                                     x,y+1)]-v0[RN(x,y)])/(DY*DY);
374                                     }
375                                     HHP_PRIME[x][y]=part1+part2;
376                                     }
377                                     }
378     }
379
380     //This represents the part of HHP that is a function of "R" and "Z" (
        NOT "PSI"). This includes any possible constant term.
381     __forceinline T f(int m, int n)
382     {
383         return 0;
384     }
385
386     #undef i
387     #undef j

```

F.2.14 ckf_null_tak.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE NULLSHEET
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      /*
69      //Unused
70      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
71      //c[#] is the coefficient for (Pij)^(#+1)
72      c[0][s][t]=0;
73      c[1][s][t]=0;
74      c[2][s][t]=0;
75      */
76      }
77  }
78
79  //The bottom right coner and bottom edge past the light
      cylinder are changed elsewhere, so the choices for these
      are largely unimportant.
80  //Left Boundary P=0
81  for(int t=1;t<=N_MAX_Y-4;t++)
82  {
83      int s=0;
84
85      bL1[t]=1;
86      bL2[t]=0;
87      bL3[t]=0;
88      bL4[t]=0;

```

```

89             bL5[t]=0;
90             bL6[t]=0;
91         }
92
93         //Right Boundary RPr+ZPz=0
94         for(int t=1;t<=N_MAX_Y-4;t++)
95         {
96             int s=N_MAX_X-3;
97
98             bR1[t]=-i;
99             bR2[t]=i;
100            bR3[t]=-j;
101            bR4[t]=j;
102            bR5[t]=0;
103            bR6[t]=0;
104        }
105
106        //Bottom Boundary
107        //Outside star, inside light cylinder Pz=0
108        for(int s=1; ((s<N_LC+1) && (s<=N_MAX_X-4)); s++)
109        {
110            int t=0;
111
112            bB1[s]=0;
113            bB2[s]=0;
114            bB3[s]=1;
115            bB4[s]=0;
116            bB5[s]=-1;
117            bB6[s]=0;
118        }
119
120        //Outside light cylinder  $H^2=(R^2-1)*(Pz)^2$ 
121        //To begin the simulation, just use  $P=P_{OP}$ 
122        for(int s=N_LC+1;s<=N_MAX_X-4;s++)
123        {
124            int t=0;
125
126            bB1[s]=0;
127            bB2[s]=0;
128            bB3[s]=1;
129            bB4[s]=0;
130            bB5[s]=0;
131            bB6[s]=P_OP;
132        }
133
134        //Top Boundary RPr+ZPz=0
135        for(int s=1;s<=N_MAX_X-4;s++)
136        {
137            int t=N_MAX_Y-3;
138
139            bT1[s]=-i;
140            bT2[s]=i;

```



```

141             bT3[s]=-j;
142             bT4[s]=j;
143             bT5[s]=0;
144             bT6[s]=0;
145         }
146
147         //Bottom Left Corner
148         {
149             int s=0;
150             int t=0;
151
152             bL1[t]=1;
153             bL2[t]=0;
154             bL3[t]=0;
155             bL4[t]=0;
156             bL5[t]=0;
157             bL6[t]=0;
158
159             bB1[s]=0;
160             bB2[s]=0;
161             bB3[s]=1;
162             bB4[s]=0;
163             bB5[s]=0;
164             bB6[s]=1;
165         }
166
167         //Top Left Corner
168         {
169             int s=0;
170             int t=N_MAX_Y-3;
171
172             bL1[t]=1;
173             bL2[t]=0;
174             bL3[t]=0;
175             bL4[t]=0;
176             bL5[t]=0;
177             bL6[t]=0;
178
179             bT1[s]=0;
180             bT2[s]=i;
181             bT3[s]=-j;
182             bT4[s]=j;
183             bT5[s]=0;
184             bT6[s]=0;
185         }
186
187         //Bottom Right Corner
188         {
189             int s=N_MAX_X-3;
190             int t=0;
191
192             bR1[t]=-i;

```

```

193         bR2[t]=i;
194         bR3[t]=0;
195         bR4[t]=j;
196         bR5[t]=0;
197         bR6[t]=j*P_OP;
198
199         bB1[s]=0;
200         bB2[s]=0;
201         bB3[s]=1;
202         bB4[s]=0;
203         bB5[s]=0;
204         bB6[s]=P_OP;
205     }
206
207     //Top Right Corner
208     {
209         int s=N_MAX_X-3;
210         int t=N_MAX_Y-3;
211
212         bR1[t]=-1;
213         bR2[t]=1;
214         bR3[t]=0;
215         bR4[t]=0;
216         bR5[t]=0;
217         bR6[t]=0;
218
219         bT1[s]=0;
220         bT2[s]=0;
221         bT3[s]=-1;
222         bT4[s]=1;
223         bT5[s]=0;
224         bT6[s]=0;
225     }
226 }
227
228 //This function inserts the star.
229 void star(void)
230 {
231     for(int s=0;s<N_S_X;s++)
232     {
233         const T R=DX*i;
234         for(int t=0;t<N_S_Y;t++)
235         {
236             const T Z=DY*j;
237             a1[s][t]=0;
238             a2[s][t]=0;
239             a3[s][t]=0;
240             a4[s][t]=0;
241             a5[s][t]=1;
242             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
243         }
244     }

```

```

245 }
246
247 //This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
248 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
249 {
250     if (toggle){printf("Takamori\n");return;}
251     printf("CKF\n");
252
253     //T Slopes[NZ_MAX+1];
254     T HHP_L[N_MAX_Y-2];
255     for(int KP=0;KP<N_MAX_Y-2;KP++)
256     {
257         HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]+v0[RN(
            N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX);
258     }
259
260     //at r<rL, there are many possible cases (closed field lines,
        open field lines that do or do not cross the light
        cylinder)
261     //for 0<r<=NR_S, we stay OUTSIDE star, because the star value
        is known and does not need to be altered.
262     for(int jj=0;jj<=N_S_X-1;jj++)
263     {
264         for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
265         {
266             int KP=0;
267             {
268                 const T PT=v0[RN(jj,kk)];
269
270                 if(PT<v0[RN(N_LC-2,0)])
271                 {
272                     if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
273                     {
274                         HHP[jj][kk]=HHP_L[
                            N_MAX_Y-3]*PT/v0[
                                RN(N_LC-2,N_MAX_Y
                                    -3)];
275                     }
276                     else
277                     {
278                         for(;KP<N_MAX_Y-4&&PT
                            <v0[RN(N_LC-2,KP)]
                                ;KP++);
279
280                         const T Q1=PT-v0[RN(
                            N_LC-1,KP+1)];
281                         const T Q2=PT-v0[RN(
                            N_LC-1,KP)];
282                         HHP[jj][kk]=(Q1*HHP_L
                            [KP]-Q2*HHP_L[KP

```

```

283                                     +1)) / (Q1-Q2);
284                                     }
285                                     else
286                                     {
287                                         HHP[jj][kk]=0;
288                                     }
289                                     }
290                                     if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
291                                 }
292                            }
293
294                            //for NR_S<r<rLC
295                            for(int jj=N_S_X;jj<=N_LC-2;jj++)
296                            {
297                                for(int kk=0;kk<=N_MAX_Y-3;kk++)
298                                {
299                                    int KP=0;
300                                    {
301                                        const T PT=v0[RN(jj,kk)];
302
303                                        if(PT<v0[RN(N_LC-2,0)])
304                                        {
305                                            if(PT<v0[RN(N_LC-2,N_MAX_Y-3)])
306                                            {
307                                                HHP[jj][kk]=HHP_L[
                                                    N_MAX_Y-3]*PT/v0[
                                                    RN(N_LC-2,N_MAX_Y
                                                    -3)];
308                                            }
309                                            else
310                                            {
311                                                for(;KP<N_MAX_Y-4&&PT
                                                    <v0[RN(N_LC-2,KP)]
                                                    ;KP++);
312
313                                                const T Q1=PT-v0[RN(
                                                    N_LC-1,KP+1)];
314                                                const T Q2=PT-v0[RN(
                                                    N_LC-1,KP)];
315                                                HHP[jj][kk]=(Q1*HHP_L
                                                    [KP]-Q2*HHP_L[KP
                                                    +1]) / (Q1-Q2);
316                                            }
317                                        }
318                                        else
319                                        {
320                                            HHP[jj][kk]=0;
321                                        }
322                                    }
323                                    if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}

```

```

324     }
325 }
326
327 for(int jj=N_LC-1;jj==N_LC-1;jj++)
328 {
329     for(int kk=0;kk<=N_MAX_Y-3;kk++)
330     {
331         HHP[N_LC-1][kk]=HHP_L[kk];
332     }
333 }
334
335 for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
336 {
337     for(int kk=0;kk<=N_MAX_Y-3;kk++)
338     {
339         int KP=0;
340         const T PT=v0[RN(jj,kk)];
341
342         for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP)];KP++)
343             ;
344
345         const T Q1=PT-v0[RN(N_LC-1,KP+1)];
346         const T Q2=PT-v0[RN(N_LC-1,KP)];
347         //if(fabs(Q1-Q2)>0.00001)
348         {
349             HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*HHP_L[KP
350                 +1])/(Q1-Q2);
351         }
352         if(isnan(HHP[jj][kk])){HHP[jj][kk]=0;}
353     }
354 }
355
356 //TODO - HHP_PRIME is largely unnecessary, and this
357 //calculation may be wrong.
358 for(int x=0;x<N_MAX_X-2;x++)
359 {
360     for(int y=0;y<N_MAX_Y-2;y++)
361     {
362         T part1,part2;
363         if(x==N_MAX_X-3)
364         {
365             part1=(HHP[x][y]-HHP[x-1][y])*(v0[RN(
366                 x,y)]-v0[RN(x-1,y)])/(DX*DX);
367         }
368         else
369         {
370             part1=(HHP[x+1][y]-HHP[x][y])*(v0[RN(
371                 x+1,y)]-v0[RN(x,y)])/(DX*DX);
372         }
373         if(y==N_MAX_Y-3)
374         {

```

```

370                                     part2=(HHP[x][y]-HHP[x][y-1])*(v0[RN(
                                     x,y)]-v0[RN(x,y-1)])/(DY*DY);
371                                     }
372                                     else
373                                     {
374                                     part2=(HHP[x][y+1]-HHP[x][y])*(v0[RN(
                                     x,y+1)]-v0[RN(x,y)])/(DY*DY);
375                                     }
376                                     HHP_PRIME[x][y]=part1+part2;
377                                     }
378                                     }
379
380                                     resetCKF();
381     }
382
383     //This represents the part of HHP that is a function of "R" and "Z" (
384     NOT "PSI"). This includes any possible constant term.
384     __forceinline T f(int m, int n)
385     {
386         return 0;
387     }
388
389     #undef i
390     #undef j

```

F.2.15 tak_ckf.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE STANDARD
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
69      //c[#] is the coefficient for (Pij)^(#+1)
70      const T A_A=1/(RATIO*P_OP*P_OP);
71      const T P_RET=RATIO*P_OP;
72
73      c[0][s][t]=A_A*P_OP*P_RET;
74      c[1][s][t]=-A_A*(P_OP+P_RET);
75      c[2][s][t]=A_A;
76      }
77  }
78
79  //Left Boundary P=0
80  for(int t=1;t<=N_MAX_Y-4;t++)
81  {
82      int s=0;
83
84      bL1[t]=1;
85      bL2[t]=0;
86      bL3[t]=0;
87      bL4[t]=0;
88      bL5[t]=0;
89      bL6[t]=0;
90  }

```



```

91
92 //Right Boundary RPr+ZPz=0
93 for(int t=1;t<=N_MAX_Y-4;t++)
94 {
95     int s=N_MAX_X-3;
96
97     bR1[t]=-i;
98     bR2[t]=i;
99     bR3[t]=-j;
100    bR4[t]=j;
101    bR5[t]=0;
102    bR6[t]=0;
103 }
104
105 //Bottom Boundary
106 //Outside star, inside light cylinder Pz=0
107 for(int s=1; ( (s<N_LC-1) && (s<=N_MAX_X-4) );s++)
108 {
109     int t=0;
110
111     bB1[s]=0;
112     bB2[s]=0;
113     bB3[s]=1;
114     bB4[s]=0;
115     bB5[s]=-1;
116     bB6[s]=0;
117 }
118
119 //Outside light cylinder P=P_OP which is specified from the
    start.
120 for(int s=N_LC-1;s<=N_MAX_X-4;s++)
121 {
122     int t=0;
123
124     bB1[s]=0;
125     bB2[s]=0;
126     bB3[s]=1;
127     bB4[s]=0;
128     bB5[s]=0;
129     bB6[s]=P_OP;
130 }
131
132 //Top Boundary RPr+ZPz=0
133 for(int s=1;s<=N_MAX_X-4;s++)
134 {
135     int t=N_MAX_Y-3;
136
137     bT1[s]=-i;
138     bT2[s]=i;
139     bT3[s]=-j;
140     bT4[s]=j;
141     bT5[s]=0;

```

```

142             bT6[s]=0;
143     }
144
145     //Bottom Left Corner
146     {
147         int s=0;
148         int t=0;
149
150         bL1[t]=1;
151         bL2[t]=0;
152         bL3[t]=0;
153         bL4[t]=0;
154         bL5[t]=0;
155         bL6[t]=0;
156
157         bB1[s]=0;
158         bB2[s]=0;
159         bB3[s]=1;
160         bB4[s]=0;
161         bB5[s]=0;
162         bB6[s]=P_OP;
163     }
164
165     //Top Left Corner
166     {
167         int s=0;
168         int t=N_MAX_Y-3;
169
170         bL1[t]=1;
171         bL2[t]=0;
172         bL3[t]=0;
173         bL4[t]=0;
174         bL5[t]=0;
175         bL6[t]=0;
176
177         bT1[s]=0;
178         bT2[s]=i;
179         bT3[s]=-j;
180         bT4[s]=j;
181         bT5[s]=0;
182         bT6[s]=0;
183     }
184
185     //Bottom Right Corner
186     {
187         int s=N_MAX_X-3;
188         int t=0;
189
190         bR1[t]=-i;
191         bR2[t]=i;
192         bR3[t]=0;
193         bR4[t]=j;

```

```

194         bR5[t]=0;
195         bR6[t]=j*P_OP;
196
197         bB1[s]=0;
198         bB2[s]=0;
199         bB3[s]=1;
200         bB4[s]=0;
201         bB5[s]=0;
202         bB6[s]=P_OP;
203     }
204
205     //Top Right Corner
206     {
207         int s=N_MAX_X-3;
208         int t=N_MAX_Y-3;
209
210         bR1[t]=-1;
211         bR2[t]=1;
212         bR3[t]=0;
213         bR4[t]=0;
214         bR5[t]=0;
215         bR6[t]=0;
216
217         bT1[s]=0;
218         bT2[s]=0;
219         bT3[s]=-1;
220         bT4[s]=1;
221         bT5[s]=0;
222         bT6[s]=0;
223     }
224 }
225
226 //This function inserts the star.
227 void star(void)
228 {
229     for(int s=0;s<N_S_X;s++)
230     {
231         const T R=DX*i;
232         for(int t=0;t<N_S_Y;t++)
233         {
234             const T Z=DY*j;
235             a1[s][t]=0;
236             a2[s][t]=0;
237             a3[s][t]=0;
238             a4[s][t]=0;
239             a5[s][t]=1;
240             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
241         }
242     }
243 }
244

```

```

245 //This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
246 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
247 {
248     if (!toggle)
249     {
250         printf("Takamori\n");
251
252         for(int x=0;x<N_MAX_X-2;x++)
253         {
254             for(int y=0;y<N_MAX_Y-2;y++)
255             {
256                 if (v0[RN(x,y)]>=P_OP)
257                 {
258                     HHP[x][y]=0;
259                     HHP_PRIME[x][y]=0;
260                 }
261                 else
262                 {
263                     HHP[x][y]=c[0][x][y]*v0[RN(x,
264                                     y)]+c[1][x][y]*v0[RN(x,y)
265                                     ]*v0[RN(x,y)]+c[2][x][y]*
266                                     v0[RN(x,y)]*v0[RN(x,y)]*v0
267                                     [RN(x,y)];
268                     HHP_PRIME[x][y]=c[0][x][y]+2*
269                                     c[1][x][y]*v0[RN(x,y)]+3*c
270                                     [2][x][y]*v0[RN(x,y)]*v0[
271                                     RN(x,y)];
272                 }
273             }
274         }
275     }
276     else
277     {
278         printf("CKF\n");
279
280         T HHP_L[N_MAX_Y-2];
281         for(int KP=0;KP<N_MAX_Y-2;KP++)
282         {
283             HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)
284                                     ]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX
285                                     );
286         }
287
288         //at r<rL, there are many possible cases (closed
289             field lines, open field lines that do or do not
290             cross the light cylinder)
291         //for 0<r<=NR_S, we stay OUTSIDE star, because the
292             star value is known and does not need to be
293             altered.
294         for(int jj=0;jj<=N_S_X-1;jj++)
295         {

```

```

283     for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
284     {
285         int KP=0;
286         {
287             const T PT=v0[RN(jj, kk)];
288
289             if (PT<v0[RN(N_LC-2, 0)])
290             {
291                 if (PT<v0[RN(N_LC-2,
292                             N_MAX_Y-3)])
293                 {
294                     HHP[jj][kk]=
295                         HHP_L[
296                             N_MAX_Y
297                             -3]*PT/v0[
298                                 RN(N_LC-2,
299                                     N_MAX_Y-3)
300                                 ];
301
302                     }
303                     else
304                     {
305                         for(;KP<
306                             N_MAX_Y
307                             -4&&PT<v0[
308                                 RN(N_LC-2,
309                                     KP)] ;KP++)
310                             ;
311
312                         const T Q1=PT
313                             -v0[RN(
314                                 N_LC-1, KP
315                                 +1)];
316                         const T Q2=PT
317                             -v0[RN(
318                                 N_LC-1, KP)
319                                 ];
320                         HHP[jj][kk]=(
321                             Q1*HHP_L[
322                                 KP]-Q2*
323                                 HHP_L[KP
324                                     +1])/ (Q1-
325                                         Q2);
326
327                     }
328                 }
329             }
330             else
331             {
332                 HHP[jj][kk]=0;
333             }
334         }
335         if (isnan(HHP[jj][kk])) {HHP[jj][kk]
336             =0;}
337     }
338 }

```

```

311     }
312
313     //for NR_S<r<rLC
314     for(int jj=N_S_X; jj<=N_LC-2; jj++)
315     {
316         for(int kk=0; kk<=N_MAX_Y-3; kk++)
317         {
318             int KP=0;
319             {
320                 const T PT=v0[RN(jj,kk)];
321
322                 if (PT<v0[RN(N_LC-2,0)])
323                 {
324                     if (PT<v0[RN(N_LC-2,
325                                     N_MAX_Y-3)])
326                     {
327                         HHP[jj][kk]=
328                             HHP_L[
329                                 N_MAX_Y
330                                 -3]*PT/v0[
331                                     RN(N_LC-2,
332                                         N_MAX_Y-3)
333                                     ];
334
335                     }
336                     else
337                     {
338                         for(; KP<
339                             N_MAX_Y
340                             -4&&PT<v0[
341                                 RN(N_LC-2,
342                                     KP)
343                                 ]; KP++)
344                         {
345                             const T Q1=PT
346                                 -v0[RN(
347                                     N_LC-1, KP
348                                     +1)];
349                             const T Q2=PT
350                                 -v0[RN(
351                                     N_LC-1, KP)
352                                     ];
353                             HHP[jj][kk]=(
354                                 Q1*HHP_L[
355                                     KP]-Q2*
356                                     HHP_L[KP
357                                         +1])/ (Q1-
358                                             Q2);
359                         }
360                     }
361                 }
362             }
363         }
364     }
365     HHP[jj][kk]=0;

```

```

340                                     }
341                                     }
342                                     if (isnan (HHP [jj] [kk])) {HHP [jj] [kk
                                     ]=0;}
343                                     }
344     }
345
346     for (int jj=N_LC-1; jj==N_LC-1; jj++)
347     {
348         for (int kk=0; kk<=N_MAX_Y-3; kk++)
349         {
350             HHP [N_LC-1] [kk]=HHP_L [kk];
351         }
352     }
353
354     for (int jj=N_LC; jj<=N_MAX_X-3; jj++)
355     {
356         for (int kk=0; kk<=N_MAX_Y-3; kk++)
357         {
358             int KP=0;
359             const T PT=v0 [RN (jj, kk) ];
360
361             for (; KP<N_MAX_Y-4&&PT<v0 [RN (N_LC-2, KP
362                                     ) ]; KP++);
363
364             const T Q1=PT-v0 [RN (N_LC-1, KP+1) ];
365             const T Q2=PT-v0 [RN (N_LC-1, KP) ];
366             //if (fabs (Q1-Q2)>0.00001)
367             {
368                 HHP [jj] [kk]=(Q1*HHP_L [KP]-Q2*
369                                     HHP_L [KP+1]) / (Q1-Q2);
370             }
371             if (isnan (HHP [jj] [kk])) {HHP [jj] [kk
372                                     ]=0;}
373
374             }
375
376     }
377
378     //TODO - HHP_PRIME is largely unnecessary, and this
379     calculation may be wrong.
380     for (int x=0; x<N_MAX_X-2; x++)
381     {
382         for (int y=0; y<N_MAX_Y-2; y++)
383         {
384             T part1, part2;
385             if (x==N_MAX_X-3)
386             {
387                 part1=(HHP [x] [y]-HHP [x-1] [y])
388                     * (v0 [RN (x, y) ]-v0 [RN (x-1, y)
389                             ] ) / (DX*DX);
390             }
391             else
392             {

```

```

385         part1=(HHP[x+1][y]-HHP[x][y])
           * (v0[RN(x+1,y)]-v0[RN(x,y)
           ]) / (DX*DX);
386     }
387     if (y==N_MAX_Y-3)
388     {
389         part2=(HHP[x][y]-HHP[x][y-1])
           * (v0[RN(x,y)]-v0[RN(x,y-1)
           ]) / (DY*DY);
390     }
391     else
392     {
393         part2=(HHP[x][y+1]-HHP[x][y])
           * (v0[RN(x,y+1)]-v0[RN(x,y)
           ]) / (DY*DY);
394     }
395     HHP_PRIME[x][y]=part1+part2;
396 }
397 }
398 }
399 }
400
401 //This represents the part of HHP that is a function of "r" and "z" (
   NOT "Phi"). This includes any possible constant term.
402 __forceinline T f(int m, int n)
403 {
404     return 0;
405 }
406
407 #undef i
408 #undef j

```


F.2.16 tak_ckf_jets.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE JETS
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
69      //c[#] is the coefficient for (Pij)^(#+1)
70      const T A_A=1/(RATIO*P_OP*P_OP);
71      const T P_RET=RATIO*P_OP;
72
73      c[0][s][t]=A_A*P_OP*P_RET;
74      c[1][s][t]=-A_A*(P_OP+P_RET);
75      c[2][s][t]=A_A;
76      }
77  }
78
79  //Left Boundary P=0
80  for(int t=1;t<=N_MAX_Y-4;t++)
81  {
82      int s=0;
83
84      bL1[t]=1;
85      bL2[t]=0;
86      bL3[t]=0;
87      bL4[t]=0;
88      bL5[t]=0;
89      bL6[t]=0;
90  }

```

```

91
92      //Right Boundary RPr+ZPz=0
93      for(int t=1;t<=N_MAX_Y-4;t++)
94      {
95          int s=N_MAX_X-3;
96
97          bR1[t]=-i;
98          bR2[t]=i;
99          bR3[t]=-j;
100         bR4[t]=j;
101         bR5[t]=0;
102         bR6[t]=0;
103     }
104
105     //Bottom Boundary
106     //Outside star, inside light cylinder Pz=0
107     for(int s=1; ( (s<N_LC-1) && (s<=N_MAX_X-4) ); s++)
108     {
109         int t=0;
110
111         bB1[s]=0;
112         bB2[s]=0;
113         bB3[s]=1;
114         bB4[s]=0;
115         bB5[s]=-1;
116         bB6[s]=0;
117     }
118
119     //Outside light cylinder P=P_OP which is specified from the
120     start.
121     for(int s=N_LC-1;s<=N_MAX_X-4;s++)
122     {
123         int t=0;
124
125         bB1[s]=0;
126         bB2[s]=0;
127         bB3[s]=1;
128         bB4[s]=0;
129         bB5[s]=0;
130         bB6[s]=P_OP;
131     }
132
133     //Top Boundary RPr+ZPz=0
134     for(int s=1;s<=N_MAX_X-4;s++)
135     {
136         int t=N_MAX_Y-3;
137
138         bT1[s]=-i;
139         bT2[s]=i;
140         bT3[s]=-j;
141         bT4[s]=j;
142         bT5[s]=0;

```

```

142             bT6[s]=0;
143     }
144
145     //Bottom Left Corner
146     {
147         int s=0;
148         int t=0;
149
150         bL1[t]=1;
151         bL2[t]=0;
152         bL3[t]=0;
153         bL4[t]=0;
154         bL5[t]=0;
155         bL6[t]=0;
156
157         bB1[s]=0;
158         bB2[s]=0;
159         bB3[s]=1;
160         bB4[s]=0;
161         bB5[s]=0;
162         bB6[s]=P_OP;
163     }
164
165     //Top Left Corner
166     {
167         int s=0;
168         int t=N_MAX_Y-3;
169
170         bL1[t]=1;
171         bL2[t]=0;
172         bL3[t]=0;
173         bL4[t]=0;
174         bL5[t]=0;
175         bL6[t]=0;
176
177         bT1[s]=0;
178         bT2[s]=i;
179         bT3[s]=-j;
180         bT4[s]=j;
181         bT5[s]=0;
182         bT6[s]=0;
183     }
184
185     //Bottom Right Corner
186     {
187         int s=N_MAX_X-3;
188         int t=0;
189
190         bR1[t]=-i;
191         bR2[t]=i;
192         bR3[t]=0;
193         bR4[t]=j;

```

```

194         bR5[t]=0;
195         bR6[t]=j*P_OP;
196
197         bB1[s]=0;
198         bB2[s]=0;
199         bB3[s]=1;
200         bB4[s]=0;
201         bB5[s]=0;
202         bB6[s]=P_OP;
203     }
204
205     //Top Right Corner
206     {
207         int s=N_MAX_X-3;
208         int t=N_MAX_Y-3;
209
210         bR1[t]=-1;
211         bR2[t]=1;
212         bR3[t]=0;
213         bR4[t]=0;
214         bR5[t]=0;
215         bR6[t]=0;
216
217         bT1[s]=0;
218         bT2[s]=0;
219         bT3[s]=-1;
220         bT4[s]=1;
221         bT5[s]=0;
222         bT6[s]=0;
223     }
224 }
225
226 //This function inserts the star.
227 void star(void)
228 {
229     for(int s=0;s<N_S_X;s++)
230     {
231         const T R=DX*i;
232         for(int t=0;t<N_S_Y;t++)
233         {
234             const T Z=DY*j;
235             a1[s][t]=0;
236             a2[s][t]=0;
237             a3[s][t]=0;
238             a4[s][t]=0;
239             a5[s][t]=1;
240             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
241         }
242     }
243 }
244

```

```

245 //This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
    and may or may not include both linear and nonlinear terms.
246 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
247 {
248     T KH=6.0;
249     T BETA=2;
250
251     if (!toggle)
252     {
253         printf("Takamori\n");
254
255         for(int x=0;x<N_MAX_X-2;x++)
256         {
257             for(int y=0;y<N_MAX_Y-2;y++)
258             {
259                 if (v0[RN(x,y)]>=P_OP)
260                 {
261                     HHP[x][y]=0;
262                     HHP_PRIME[x][y]=0;
263                 }
264                 else
265                 {
266                     HHP[x][y]=KH*KH*0.5*v0[RN(x,y)
                ]*(BETA*v0[RN(x,y)]/P_OP
                -1)*(BETA*v0[RN(x,y)]/P_OP
                -2);
267                 }
268             }
269         }
270     }
271     else
272     {
273         printf("CKF\n");
274
275         T HHP_L[N_MAX_Y-2];
276         for(int KP=0;KP<N_MAX_Y-2;KP++)
277         {
278             HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)
                ]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX
                );
279         }
280
281         //at r<rL, there are many possible cases (closed
                field lines, open field lines that do or do not
                cross the light cylinder)
282         //for 0<r<=NR_S, we stay OUTSIDE star, because the
                star value is known and does not need to be
                altered.
283         for(int jj=0;jj<=N_S_X-1;jj++)
284         {
285             for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
286             {

```

```

287      int KP=0;
288      {
289          const T PT=v0[RN(jj, kk)];
290
291          if (PT<v0[RN(N_LC-2, 0)])
292          {
293              if (PT<v0[RN(N_LC-2,
294                          N_MAX_Y-3)])
295              {
296                  HHP[jj][kk]=
297                      KH*KH*0.5*
298                      v0[RN(jj,
299                          kk)]*(BETA
300                          *v0[RN(jj,
301                          kk)]/P_OP
302                          -1)*(BETA*
303                          v0[RN(jj,
304                          kk)]/P_OP
305                          -2);
306              }
307              else
308              {
309                  for (; KP<
310                      N_MAX_Y
311                      -4&&PT<v0[
312                      RN(N_LC-2,
313                      KP)]; KP++)
314                      ;
315
316                  const T Q1=PT
317                      -v0[RN(
318                      N_LC-1, KP
319                      +1)];
320                  const T Q2=PT
321                      -v0[RN(
322                      N_LC-1, KP)
323                      ];
324                  HHP[jj][kk]=(
325                      Q1*HHP_L[
326                      KP]-Q2*
327                      HHP_L[KP
328                      +1])/(Q1-
329                      Q2);
330              }
331          }
332          else
333          {
334              HHP[jj][kk]=0;
335          }
336      }
337      if (isnan(HHP[jj][kk])) {HHP[jj][kk]
338          =0;}

```

```

312         }
313     }
314
315     //for NR_S<r<rLC
316     for(int jj=N_S_X;jj<=N_LC-2;jj++)
317     {
318         for(int kk=0;kk<=N_MAX_Y-3;kk++)
319         {
320             int KP=0;
321             {
322                 const T PT=v0[RN(jj,kk)];
323
324                 if (PT<v0[RN(N_LC-2,0)])
325                 {
326                     if (PT<v0[RN(N_LC-2,
327                                     N_MAX_Y-3)])
328                     {
329                         HHP[jj][kk]=
330                             KH*KH*0.5*
331                             v0[RN(jj,
332                                     kk)]*(BETA
333                                     *v0[RN(jj,
334                                             kk)]/P_OP
335                                     -1)*(BETA*
336                                     v0[RN(jj,
337                                             kk)]/P_OP
338                                     -2);
339                     }
340                     else
341                     {
342                         for(;KP<
343                             N_MAX_Y
344                             -4&&PT<v0[
345                                 RN(N_LC-2,
346                                     KP)];KP++)
347                             ;
348
349                         const T Q1=PT
350                             -v0[RN(
351                                 N_LC-1,KP
352                                 +1)];
353                         const T Q2=PT
354                             -v0[RN(
355                                 N_LC-1,KP)
356                                 ];
357                         HHP[jj][kk]=(
358                             Q1*HHP_L[
359                                 KP]-Q2*
360                                 HHP_L[KP
361                                     +1])/ (Q1-
362                                     Q2);
363                     }
364                 }
365             }
366         }
367     }

```



```

338         }
339         else
340         {
341             HHP[jj][kk]=0;
342         }
343     }
344     if(isnan(HHP[jj][kk])){HHP[jj][kk]
        ]=0;}
345     }
346 }
347
348 for(int jj=N_LC-1;jj==N_LC-1;jj++)
349 {
350     for(int kk=0;kk<=N_MAX_Y-3;kk++)
351     {
352         HHP[N_LC-1][kk]=HHP_L[kk];
353     }
354 }
355
356 for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
357 {
358     for(int kk=0;kk<=N_MAX_Y-3;kk++)
359     {
360         int KP=0;
361         const T PT=v0[RN(jj,kk)];
362
363         for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP
            )];KP++);
364
365         const T Q1=PT-v0[RN(N_LC-1,KP+1)];
366         const T Q2=PT-v0[RN(N_LC-1,KP)];
367         //if(fabs(Q1-Q2)>0.00001)
368         {
369             HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*
                HHP_L[KP+1])/(Q1-Q2);
370         }
371         if(isnan(HHP[jj][kk])){HHP[jj][kk]
            ]=0;}
372     }
373 }
374
375 //TODO - HHP_PRIME is largely unnecessary, and this
        calculation may be wrong.
376 for(int x=0;x<N_MAX_X-2;x++)
377 {
378     for(int y=0;y<N_MAX_Y-2;y++)
379     {
380         T part1,part2;
381         if(x==N_MAX_X-3)
382         {
383             part1=(HHP[x][y]-HHP[x-1][y])
                *(v0[RN(x,y)]-v0[RN(x-1,y)]

```

```

384                                     ]) / (DX*DX);
385                                     }
386                                     else
387                                     {
388                                         part1=(HHP[x+1][y]-HHP[x][y])
389                                             * (v0[RN(x+1,y)]-v0[RN(x,y)
390                                                 ]) / (DX*DX);
391                                     }
392                                     if (y==N_MAX_Y-3)
393                                     {
394                                         part2=(HHP[x][y]-HHP[x][y-1])
395                                             * (v0[RN(x,y)]-v0[RN(x,y-1)
396                                                 ]) / (DY*DY);
397                                     }
398                                     else
399                                     {
400                                         part2=(HHP[x][y+1]-HHP[x][y])
401                                             * (v0[RN(x,y+1)]-v0[RN(x,y)
402                                                 ]) / (DY*DY);
403                                     }
404                                     HHP_PRIME[x][y]=part1+part2;
405                                     }
406                                     }
407                                     }
408                                     }
409                                     }
410                                     }
411                                     }
412                                     }
413                                     }
414                                     }
415                                     }
416                                     }
417                                     }
418                                     }
419                                     }
420                                     }
421                                     }
422                                     }
423                                     }
424                                     }
425                                     }
426                                     }
427                                     }
428                                     }
429                                     }
430                                     }
431                                     }
432                                     }
433                                     }
434                                     }
435                                     }
436                                     }
437                                     }
438                                     }
439                                     }
440                                     }
441                                     }
442                                     }
443                                     }
444                                     }
445                                     }
446                                     }
447                                     }
448                                     }
449                                     }
450                                     }
451                                     }
452                                     }
453                                     }
454                                     }
455                                     }
456                                     }
457                                     }
458                                     }
459                                     }
460                                     }
461                                     }
462                                     }
463                                     }
464                                     }
465                                     }
466                                     }
467                                     }
468                                     }
469                                     }
470                                     }
471                                     }
472                                     }
473                                     }
474                                     }
475                                     }
476                                     }
477                                     }
478                                     }
479                                     }
480                                     }
481                                     }
482                                     }
483                                     }
484                                     }
485                                     }
486                                     }
487                                     }
488                                     }
489                                     }
490                                     }
491                                     }
492                                     }
493                                     }
494                                     }
495                                     }
496                                     }
497                                     }
498                                     }
499                                     }
500                                     }
501                                     }
502                                     }
503                                     }
504                                     }
505                                     }
506                                     }
507                                     }
508                                     }
509                                     }
510                                     }
511                                     }
512                                     }
513                                     }
514                                     }
515                                     }
516                                     }
517                                     }
518                                     }
519                                     }
520                                     }
521                                     }
522                                     }
523                                     }
524                                     }
525                                     }
526                                     }
527                                     }
528                                     }
529                                     }
530                                     }
531                                     }
532                                     }
533                                     }
534                                     }
535                                     }
536                                     }
537                                     }
538                                     }
539                                     }
540                                     }
541                                     }
542                                     }
543                                     }
544                                     }
545                                     }
546                                     }
547                                     }
548                                     }
549                                     }
550                                     }
551                                     }
552                                     }
553                                     }
554                                     }
555                                     }
556                                     }
557                                     }
558                                     }
559                                     }
560                                     }
561                                     }
562                                     }
563                                     }
564                                     }
565                                     }
566                                     }
567                                     }
568                                     }
569                                     }
570                                     }
571                                     }
572                                     }
573                                     }
574                                     }
575                                     }
576                                     }
577                                     }
578                                     }
579                                     }
580                                     }
581                                     }
582                                     }
583                                     }
584                                     }
585                                     }
586                                     }
587                                     }
588                                     }
589                                     }
590                                     }
591                                     }
592                                     }
593                                     }
594                                     }
595                                     }
596                                     }
597                                     }
598                                     }
599                                     }
600                                     }
601                                     }
602                                     }
603                                     }
604                                     }
605                                     }
606                                     }
607                                     }
608                                     }
609                                     }
610                                     }
611                                     }
612                                     }
613                                     }
614                                     }
615                                     }
616                                     }
617                                     }
618                                     }
619                                     }
620                                     }
621                                     }
622                                     }
623                                     }
624                                     }
625                                     }
626                                     }
627                                     }
628                                     }
629                                     }
630                                     }
631                                     }
632                                     }
633                                     }
634                                     }
635                                     }
636                                     }
637                                     }
638                                     }
639                                     }
640                                     }
641                                     }
642                                     }
643                                     }
644                                     }
645                                     }
646                                     }
647                                     }
648                                     }
649                                     }
650                                     }
651                                     }
652                                     }
653                                     }
654                                     }
655                                     }
656                                     }
657                                     }
658                                     }
659                                     }
660                                     }
661                                     }
662                                     }
663                                     }
664                                     }
665                                     }
666                                     }
667                                     }
668                                     }
669                                     }
670                                     }
671                                     }
672                                     }
673                                     }
674                                     }
675                                     }
676                                     }
677                                     }
678                                     }
679                                     }
680                                     }
681                                     }
682                                     }
683                                     }
684                                     }
685                                     }
686                                     }
687                                     }
688                                     }
689                                     }
690                                     }
691                                     }
692                                     }
693                                     }
694                                     }
695                                     }
696                                     }
697                                     }
698                                     }
699                                     }
700                                     }
701                                     }
702                                     }
703                                     }
704                                     }
705                                     }
706                                     }
707                                     }
708                                     }
709                                     }
710                                     }
711                                     }
712                                     }
713                                     }
714                                     }
715                                     }
716                                     }
717                                     }
718                                     }
719                                     }
720                                     }
721                                     }
722                                     }
723                                     }
724                                     }
725                                     }
726                                     }
727                                     }
728                                     }
729                                     }
730                                     }
731                                     }
732                                     }
733                                     }
734                                     }
735                                     }
736                                     }
737                                     }
738                                     }
739                                     }
740                                     }
741                                     }
742                                     }
743                                     }
744                                     }
745                                     }
746                                     }
747                                     }
748                                     }
749                                     }
750                                     }
751                                     }
752                                     }
753                                     }
754                                     }
755                                     }
756                                     }
757                                     }
758                                     }
759                                     }
760                                     }
761                                     }
762                                     }
763                                     }
764                                     }
765                                     }
766                                     }
767                                     }
768                                     }
769                                     }
770                                     }
771                                     }
772                                     }
773                                     }
774                                     }
775                                     }
776                                     }
777                                     }
778                                     }
779                                     }
780                                     }
781                                     }
782                                     }
783                                     }
784                                     }
785                                     }
786                                     }
787                                     }
788                                     }
789                                     }
790                                     }
791                                     }
792                                     }
793                                     }
794                                     }
795                                     }
796                                     }
797                                     }
798                                     }
799                                     }
800                                     }
801                                     }
802                                     }
803                                     }
804                                     }
805                                     }
806                                     }
807                                     }
808                                     }
809                                     }
810                                     }
811                                     }
812                                     }
813                                     }
814                                     }
815                                     }
816                                     }
817                                     }
818                                     }
819                                     }
820                                     }
821                                     }
822                                     }
823                                     }
824                                     }
825                                     }
826                                     }
827                                     }
828                                     }
829                                     }
830                                     }
831                                     }
832                                     }
833                                     }
834                                     }
835                                     }
836                                     }
837                                     }
838                                     }
839                                     }
840                                     }
841                                     }
842                                     }
843                                     }
844                                     }
845                                     }
846                                     }
847                                     }
848                                     }
849                                     }
850                                     }
851                                     }
852                                     }
853                                     }
854                                     }
855                                     }
856                                     }
857                                     }
858                                     }
859                                     }
860                                     }
861                                     }
862                                     }
863                                     }
864                                     }
865                                     }
866                                     }
867                                     }
868                                     }
869                                     }
870                                     }
871                                     }
872                                     }
873                                     }
874                                     }
875                                     }
876                                     }
877                                     }
878                                     }
879                                     }
880                                     }
881                                     }
882                                     }
883                                     }
884                                     }
885                                     }
886                                     }
887                                     }
888                                     }
889                                     }
890                                     }
891                                     }
892                                     }
893                                     }
894                                     }
895                                     }
896                                     }
897                                     }
898                                     }
899                                     }
900                                     }
901                                     }
902                                     }
903                                     }
904                                     }
905                                     }
906                                     }
907                                     }
908                                     }
909                                     }
910                                     }
911                                     }
912                                     }
913                                     }
914                                     }
915                                     }
916                                     }
917                                     }
918                                     }
919                                     }
920                                     }
921                                     }
922                                     }
923                                     }
924                                     }
925                                     }
926                                     }
927                                     }
928                                     }
929                                     }
930                                     }
931                                     }
932                                     }
933                                     }
934                                     }
935                                     }
936                                     }
937                                     }
938                                     }
939                                     }
940                                     }
941                                     }
942                                     }
943                                     }
944                                     }
945                                     }
946                                     }
947                                     }
948                                     }
949                                     }
950                                     }
951                                     }
952                                     }
953                                     }
954                                     }
955                                     }
956                                     }
957                                     }
958                                     }
959                                     }
960                                     }
961                                     }
962                                     }
963                                     }
964                                     }
965                                     }
966                                     }
967                                     }
968                                     }
969                                     }
970                                     }
971                                     }
972                                     }
973                                     }
974                                     }
975                                     }
976                                     }
977                                     }
978                                     }
979                                     }
980                                     }
981                                     }
982                                     }
983                                     }
984                                     }
985                                     }
986                                     }
987                                     }
988                                     }
989                                     }
990                                     }
991                                     }
992                                     }
993                                     }
994                                     }
995                                     }
996                                     }
997                                     }
998                                     }
999                                     }
1000                                    }

```

403 *//This represents the part of HHP that is a function of "R" and "Z" (*
404 *NOT "PSI"). This includes any possible constant term.*
405 `__forceinline T f(int m, int n)`
406 `{`
407 `return 0;`
408 `}`
409 `#undef i`
410 `#undef j`

F.2.17 tak_ckf_null.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE NULLSHEET
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68      /*
69      //Unused
70      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
71      //c[#] is the coefficient for (Pij)^(#+1)
72      const T A_A=1/(RATIO*P_OP*P_OP);
73      const T P_RET=RATIO*P_OP;
74
75      c[0][s][t]=A_A*P_OP*P_RET;
76      c[1][s][t]=-A_A*(P_OP+P_RET);
77      c[2][s][t]=A_A;
78      */
79      }
80  }
81
82  //The bottom right coner and bottom edge past the light
      cylinder are changed elsewhere, so the choices for these
      are largely unimportant.
83  //Left Boundary P=0
84  for(int t=1;t<=N_MAX_Y-4;t++)
85  {
86      int s=0;
87
88      bL1[t]=1;

```

```

89             bL2[t]=0;
90             bL3[t]=0;
91             bL4[t]=0;
92             bL5[t]=0;
93             bL6[t]=0;
94     }
95
96     //Right Boundary RPr+ZPz=0
97     for(int t=1;t<=N_MAX_Y-4;t++)
98     {
99         int s=N_MAX_X-3;
100
101         bR1[t]=-i;
102         bR2[t]=i;
103         bR3[t]=-j;
104         bR4[t]=j;
105         bR5[t]=0;
106         bR6[t]=0;
107     }
108
109     //Bottom Boundary
110     //Outside star, inside light cylinder Pz=0
111     for(int s=1; ((s<N_LC-1) && (s<=N_MAX_X-4)); s++)
112     {
113         int t=0;
114
115         bB1[s]=0;
116         bB2[s]=0;
117         bB3[s]=1;
118         bB4[s]=0;
119         bB5[s]=-1;
120         bB6[s]=0;
121     }
122
123     //Outside light cylinder  $H^2=(R^2-1)*(Pz)^2$ 
124     //To begin the simulation, just use  $P=P_{OP}$ 
125     for(int s=N_LC-1; s<=N_MAX_X-4; s++)
126     {
127         int t=0;
128
129         bB1[s]=0;
130         bB2[s]=0;
131         bB3[s]=1;
132         bB4[s]=0;
133         bB5[s]=0;
134         bB6[s]=P_OP;
135     }
136
137     //Top Boundary RPr+ZPz=0
138     for(int s=1; s<=N_MAX_X-4; s++)
139     {
140         int t=N_MAX_Y-3;

```

```

141
142             bT1[s]=-i;
143             bT2[s]=i;
144             bT3[s]=-j;
145             bT4[s]=j;
146             bT5[s]=0;
147             bT6[s]=0;
148     }
149
150     //Bottom Left Corner
151     {
152         int s=0;
153         int t=0;
154
155         bL1[t]=1;
156         bL2[t]=0;
157         bL3[t]=0;
158         bL4[t]=0;
159         bL5[t]=0;
160         bL6[t]=0;
161
162         bB1[s]=0;
163         bB2[s]=0;
164         bB3[s]=1;
165         bB4[s]=0;
166         bB5[s]=0;
167         bB6[s]=P_OP;
168     }
169
170     //Top Left Corner
171     {
172         int s=0;
173         int t=N_MAX_Y-3;
174
175         bL1[t]=1;
176         bL2[t]=0;
177         bL3[t]=0;
178         bL4[t]=0;
179         bL5[t]=0;
180         bL6[t]=0;
181
182         bT1[s]=0;
183         bT2[s]=i;
184         bT3[s]=-j;
185         bT4[s]=j;
186         bT5[s]=0;
187         bT6[s]=0;
188     }
189
190     //Bottom Right Corner
191     {
192         int s=N_MAX_X-3;

```

```

193         int t=0;
194
195         bR1[t]=-i;
196         bR2[t]=i;
197         bR3[t]=0;
198         bR4[t]=j;
199         bR5[t]=0;
200         bR6[t]=j*P_OP;
201
202         bB1[s]=0;
203         bB2[s]=0;
204         bB3[s]=1;
205         bB4[s]=0;
206         bB5[s]=0;
207         bB6[s]=P_OP;
208     }
209
210     //Top Right Corner
211     {
212         int s=N_MAX_X-3;
213         int t=N_MAX_Y-3;
214
215         bR1[t]=-1;
216         bR2[t]=1;
217         bR3[t]=0;
218         bR4[t]=0;
219         bR5[t]=0;
220         bR6[t]=0;
221
222         bT1[s]=0;
223         bT2[s]=0;
224         bT3[s]=-1;
225         bT4[s]=1;
226         bT5[s]=0;
227         bT6[s]=0;
228     }
229 }
230
231 //This function inserts the star.
232 void star(void)
233 {
234     for(int s=0;s<N_S_X;s++)
235     {
236         const T R=DX*i;
237         for(int t=0;t<N_S_Y;t++)
238         {
239             const T Z=DY*j;
240             a1[s][t]=0;
241             a2[s][t]=0;
242             a3[s][t]=0;
243             a4[s][t]=0;
244             a5[s][t]=1;

```

```

245             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
246         }
247     }
248 }
249
250 //This fills in HHP and d(HHP)/d(PSI), which is a function of "PSI"
251 and may or may not include both linear and nonlinear terms.
252 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
253 {
254     if (!toggle)
255     {
256         printf("Takamori\n");
257         for(int x=0;x<N_MAX_X-2;x++)
258         {
259             for(int y=0;y<N_MAX_Y-2;y++)
260             {
261                 if(v0[RN(x,y)]>=1)
262                 {
263                     HHP[x][y]=0;
264                 }
265                 else
266                 {
267                     HHP[x][y]=(1.07*v0[RN(x,y)]
268                                *(2-v0[RN(x,y)])*pow(1-v0
269                                [RN(x,y)],0.4))
270                                *(0.428*(5+6*v0[RN(x,y)]*(
271                                v0[RN(x,y)]-2))/(pow(1-v0[
272                                RN(x,y)],0.6)));
273                 }
274             }
275         }
276         resetTak();
277     }
278     else
279     {
280         printf("CKF\n");
281         T HHP_L[N_MAX_Y-2];
282         for(int KP=0;KP<N_MAX_Y-2;KP++)
283         {
284             HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)]
285                        +v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX
286                        );
287         }
288         //at r<rL, there are many possible cases (closed
289         field lines, open field lines that do or do not
290         cross the light cylinder)
291         //for 0<r<=NR_S, we stay OUTSIDE star, because the
292         star value is known and does not need to be
293         altered.

```



```

285     for(int jj=0;jj<=N_S_X-1;jj++)
286     {
287         for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
288         {
289             int KP=0;
290             {
291                 const T PT=v0[RN(jj,kk)];
292
293                 if (PT<v0[RN(N_LC-2,0)])
294                 {
295                     if (PT<v0[RN(N_LC-2,
296                                     N_MAX_Y-3)])
297                     {
298                         HHP[jj][kk]=
299                         HHP_L[
300                         N_MAX_Y
301                         -3]*PT/v0[
302                         RN(N_LC-2,
303                         N_MAX_Y-3)
304                         ];
305
306                     }
307                     else
308                     {
309                         for (;KP<
310                             N_MAX_Y
311                             -4&&PT<v0[
312                             RN(N_LC-2,
313                                 KP)];KP++)
314                             ;
315
316                         const T Q1=PT
317                         -v0[RN(
318                             N_LC-1,KP
319                             +1)];
320
321                         const T Q2=PT
322                         -v0[RN(
323                             N_LC-1,KP)
324                         ];
325
326                         HHP[jj][kk]=(
327                             Q1*HHP_L[
328                             KP]-Q2*
329                             HHP_L[KP
330                             +1])/ (Q1-
331                             Q2);
332
333                     }
334                 }
335             }
336
337             HHP[jj][kk]=0;
338         }
339     }
340 }

```

```

313         if (isnan (HHP [jj] [kk])) {HHP [jj] [kk
314             ]=0;}
315     }
316 }
317 //for NR_S<r<rLC
318 for (int jj=N_S_X; jj<=N_LC-2; jj++)
319 {
320     for (int kk=0; kk<=N_MAX_Y-3; kk++)
321     {
322         int KP=0;
323         {
324             const T PT=v0 [RN (jj, kk)];
325
326             if (PT<v0 [RN (N_LC-2, 0)])
327             {
328                 if (PT<v0 [RN (N_LC-2,
329                     N_MAX_Y-3)])
330                 {
331                     HHP [jj] [kk]=
332                         HHP_L[
333                             N_MAX_Y
334                             -3]*PT/v0 [
335                                 RN (N_LC-2,
336                                     N_MAX_Y-3)
337                                 ];
338                 }
339                 else
340                 {
341                     for (; KP<
342                         N_MAX_Y
343                         -4&&PT<v0 [
344                             RN (N_LC-2,
345                                 KP) ]; KP++)
346                     ;
347
348                     const T Q1=PT
349                         -v0 [RN (
350                             N_LC-1, KP
351                             +1)];
352                     const T Q2=PT
353                         -v0 [RN (
354                             N_LC-1, KP)
355                         ];
356                     HHP [jj] [kk]= (
357                         Q1*HHP_L[
358                             KP]-Q2*
359                             HHP_L[KP
360                             +1])/ (Q1-
361                             Q2);
362                 }
363             }
364         }
365     }
366 }

```

```

341                                     else
342                                     {
343                                         HHP[jj][kk]=0;
344                                     }
345                                 }
346                                 if(isnan(HHP[jj][kk])){HHP[jj][kk]
                                     ]=0;}
347                             }
348                         }
349
350                     for(int jj=N_LC-1;jj==N_LC-1;jj++)
351                     {
352                         for(int kk=0;kk<=N_MAX_Y-3;kk++)
353                         {
354                             HHP[N_LC-1][kk]=HHP_L[kk];
355                         }
356                     }
357
358                     for(int jj=N_LC;jj<=N_MAX_X-3;jj++)
359                     {
360                         for(int kk=0;kk<=N_MAX_Y-3;kk++)
361                         {
362                             int KP=0;
363                             const T PT=v0[RN(jj,kk)];
364
365                             for(;KP<N_MAX_Y-4&&PT<v0[RN(N_LC-2,KP
                                     )];KP++);
366
367                             const T Q1=PT-v0[RN(N_LC-1,KP+1)];
368                             const T Q2=PT-v0[RN(N_LC-1,KP)];
369                             //if(fabs(Q1-Q2)>0.00001)
370                             {
371                                 HHP[jj][kk]=(Q1*HHP_L[KP]-Q2*
                                     HHP_L[KP+1])/(Q1-Q2);
372                             }
373                             if(isnan(HHP[jj][kk])){HHP[jj][kk]
                                     ]=0;}
374                         }
375                     }
376
377                     //TODO - HHP_PRIME is largely unnecessary, and this
                           calculation may be wrong.
378                     for(int x=0;x<N_MAX_X-2;x++)
379                     {
380                         for(int y=0;y<N_MAX_Y-2;y++)
381                         {
382                             T part1,part2;
383                             if(x==N_MAX_X-3)
384                             {
385                                 part1=(HHP[x][y]-HHP[x-1][y])
                                     *(v0[RN(x,y)]-v0[RN(x-1,y)
                                     ])/(DX*DX);

```

```

386     }
387     else
388     {
389         part1=(HHP[x+1][y]-HHP[x][y])
            * (v0[RN(x+1,y)]-v0[RN(x,y)
            ]) / (DX*DX);
390     }
391     if(y==N_MAX_Y-3)
392     {
393         part2=(HHP[x][y]-HHP[x][y-1])
            * (v0[RN(x,y)]-v0[RN(x,y-1)
            ]) / (DY*DY);
394     }
395     else
396     {
397         part2=(HHP[x][y+1]-HHP[x][y])
            * (v0[RN(x,y+1)]-v0[RN(x,y)
            ]) / (DY*DY);
398     }
399     HHP_PRIME[x][y]=part1+part2;
400     }
401     }
402     resetCKF();
403 }
404 }
405
406 //This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
407 __forceinline T f(int m, int n)
408 {
409     return 0;
410 }
411
412 #undef i
413 #undef j

```

F.2.18 tak_theory_jets_ckf.c

```
1  /*
2  Refers to the type of simulation.
3  If it is a double simulation, this must match whatever the end graph
   will be of.
4  If there is more than one type of simulation mixed together, this
   must match whatever the bottom boundary is of.
5  */
6  #define TYPE JETS
7  #define i ((T)(s+1))
8  #define j ((T)(t+1))
9
10 void initialize(void)
11 {
12     //Equation coefficients
13     for(int s=0;s<=N_MAX_X-3;s++)
14     {
15         for(int t=0;t<=N_MAX_Y-3;t++)
16         {
17             /*
18             //These equations represent an alternate
19             finite difference choice for the first
20             derivative.
21             //The upper right corner is impossible to
22             solve for using this choice, so I don't
23             recommend this option.
24             //1st derivative is central difference
25             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
26             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
27             a3[s][t]=1-i*i*DX*DX;
28             a4[s][t]=1-i*i*DX*DX;
29             a5[s][t]=-4+4*i*i*DX*DX;
30             a6[s][t]=0;
31             */
32             /*
33             //1st derivative is central difference, DX
34             and DY independent
35             a1[s][t]=1-i*i*DX*DX+0.5*(1/i+i*DX*DX);
36             a2[s][t]=1-i*i*DX*DX-0.5*(1/i+i*DX*DX);
37             a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
38             a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
39             a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
40             ;
41             a6[s][t]=0;
42             */
43             /*
44             //1st derivative is backward difference
45             a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
46             a2[s][t]=1-i*i*DX*DX;
```

```

43      a3[s][t]=1-i*i*DX*DX;
44      a4[s][t]=1-i*i*DX*DX;
45      a5[s][t]=-4+4*i*i*DX*DX-1/i-i*DX*DX;
46      a6[s][t]=0;
47      */
48
49      //1st derivative is backward difference, DX
      and DY independent
50      a1[s][t]=1-i*i*DX*DX+1/i+i*DX*DX;
51      a2[s][t]=1-i*i*DX*DX;
52      a3[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
53      a4[s][t]=(DX*DX/(DY*DY))*(1-i*i*DX*DX);
54      a5[s][t]=(-2-2*(DX*DX/(DY*DY)))*(1-i*i*DX*DX)
      -1/i-i*DX*DX;
55      a6[s][t]=0;
56
57      //Value near LC is average of values to the
      left and right.
58      if (s>=N_LC-1-SMOOTH&&s<=N_LC-1+SMOOTH)
59      {
60          a1[s][t]=-0.5;
61          a2[s][t]=-0.5;
62          a3[s][t]=0;
63          a4[s][t]=0;
64          a5[s][t]=1;
65          a6[s][t]=0;
66      }
67
68
69      //Polynomial coefficients of HHP (constant
      term taken care of in function "f")
70      //c[#] is the coefficient for (Pij)^(#+1)
71      const T A_A=1/(RATIO*P_OP*P_OP);
72      const T P_RET=RATIO*P_OP;
73
74      c[0][s][t]=A_A*P_OP*P_RET;
75      c[1][s][t]=-A_A*(P_OP+P_RET);
76      c[2][s][t]=A_A;
77      }
78  }
79
80  //Left Boundary P=0
81  for(int t=1;t<=N_MAX_Y-4;t++)
82  {
83      int s=0;
84
85      bL1[t]=1;
86      bL2[t]=0;
87      bL3[t]=0;
88      bL4[t]=0;
89      bL5[t]=0;
90      bL6[t]=0;

```

```

91     }
92
93     //Right Boundary P=P_OP
94     for(int t=1;t<=N_MAX_Y-4;t++)
95     {
96         int s=N_MAX_X-3;
97
98         bR1[t]=0;
99         bR2[t]=1;
100        bR3[t]=0;
101        bR4[t]=0;
102        bR5[t]=0;
103        bR6[t]=P_OP;
104    }
105
106    //Bottom Boundary P=P_OP
107    for(int s=1;s<=N_MAX_X-4;s++)
108    {
109        int t=0;
110
111        bB1[s]=0;
112        bB2[s]=0;
113        bB3[s]=1;
114        bB4[s]=0;
115        bB5[s]=0;
116        bB6[s]=P_OP;
117    }
118
119    //Top Boundary
120    //Inside light cylinder Pz=0
121    for(int s=1; ( (s<N_LC-1) && (s<=N_MAX_X-4) ); s++)
122    {
123        int t=N_MAX_Y-3;
124
125        bT1[s]=0;
126        bT2[s]=0;
127        bT3[s]=0;
128        bT4[s]=1;
129        bT5[s]=-1;
130        bT6[s]=0;
131    }
132
133    //Outside light cylinder P=P_OP which is specified from the
start.
134    for(int s=N_LC-1;s<=N_MAX_X-4;s++)
135    {
136        int t=N_MAX_Y-3;
137
138        bT1[s]=0;
139        bT2[s]=0;
140        bT3[s]=0;
141        bT4[s]=1;

```

```

142             bT5[s]=0;
143             bT6[s]=P_OP;
144         }
145
146         //Bottom Left Corner
147         {
148             int s=0;
149             int t=0;
150
151             bL1[t]=1;
152             bL2[t]=0;
153             bL3[t]=0;
154             bL4[t]=0;
155             bL5[t]=0;
156             bL6[t]=0;
157
158             bB1[s]=0;
159             bB2[s]=0;
160             bB3[s]=1;
161             bB4[s]=0;
162             bB5[s]=0;
163             bB6[s]=P_OP;
164         }
165
166         //Top Left Corner
167         {
168             int s=0;
169             int t=N_MAX_Y-3;
170
171             bL1[t]=1;
172             bL2[t]=0;
173             bL3[t]=0;
174             bL4[t]=0;
175             bL5[t]=0;
176             bL6[t]=0;
177
178             bT1[s]=0;
179             bT2[s]=0;
180             bT3[s]=0;
181             bT4[s]=1;
182             bT5[s]=-1;
183             bT6[s]=0;
184         }
185
186         //Bottom Right Corner
187         {
188             int s=N_MAX_X-3;
189             int t=0;
190
191             bR1[t]=0;
192             bR2[t]=1;
193             bR3[t]=0;

```



```

194         bR4[t]=0;
195         bR5[t]=0;
196         bR6[t]=P_OP;
197
198         bB1[s]=0;
199         bB2[s]=0;
200         bB3[s]=1;
201         bB4[s]=0;
202         bB5[s]=0;
203         bB6[s]=P_OP;
204     }
205
206     //Top Right Corner
207     {
208         int s=N_MAX_X-3;
209         int t=N_MAX_Y-3;
210
211         bR1[t]=0;
212         bR2[t]=1;
213         bR3[t]=0;
214         bR4[t]=0;
215         bR5[t]=0;
216         bR6[t]=P_OP;
217
218         bT1[s]=0;
219         bT2[s]=0;
220         bT3[s]=0;
221         bT4[s]=1;
222         bT5[s]=0;
223         bT6[s]=P_OP;
224     }
225 }
226
227 //This function inserts the star.
228 void star(void)
229 {
230     for(int s=0;s<N_S_X;s++)
231     {
232         const T R=DX*i;
233         for(int t=0;t<N_S_Y;t++)
234         {
235             const T Z=DY*j;
236             a1[s][t]=0;
237             a2[s][t]=0;
238             a3[s][t]=0;
239             a4[s][t]=0;
240             a5[s][t]=1;
241             a6[s][t]=R*R/pow(R*R+Z*Z,1.5);
242         }
243     }
244 }
245

```

```

246 //This fills in HHP and d(HHP)/d(Psi), which is a function of "Psi"
    and may or may not include both linear and nonlinear terms.
247 __forceinline void hhpSet(T** HHP,T** HHP_PRIME)
248 {
249     T KH=6.6;
250     if (!toggle)
251     {
252         printf("Takamori\n");
253         for(int x=0;x<N_MAX_X-2;x++)
254         {
255             for(int y=0;y<N_MAX_Y-2;y++)
256             {
257                 if (v0[RN(x,y)]>=P_OP)
258                 {
259                     HHP[x][y]=0;
260                     HHP_PRIME[x][y]=0;
261                 }
262                 else if (v0[RN(x,y)]<=0.0){HHP[x][y]
                    ]=0;v0[RN(x,y)]=0.0;}
263                 else
264                 {
265                     T BETA=1+(KH-2)*log(fabs(v0[
                        RN(x,y)]/P_OP));
266                     HHP[x][y]=KH*KH*0.5*v0[RN(x,y)
                        ]*(BETA*v0[RN(x,y)]/P_OP
                        -1)*(BETA*v0[RN(x,y)]/P_OP
                        -2);
267                 }
268             }
269         }
270     }
271     else
272     {
273         printf("CKF\n");
274         T HHP_L[N_MAX_Y-2];
275         for(int KP=0;KP<N_MAX_Y-2;KP++)
276         {
277             HHP_L[KP]=(v0[RN(N_LC-2,KP)]-v0[RN(N_LC-3,KP)
                ]+v0[RN(N_LC+1,KP)]-v0[RN(N_LC,KP)])*(1/DX
                );
278         }
279
280         //at r<rL, there are many possible cases (closed
            field lines, open field lines that do or do not
            cross the light cylinder)
281         //for 0<r<=NR_S, we stay OUTSIDE star, because the
            star value is known and does not need to be
            altered.
282         for(int jj=0;jj<=N_S_X-1;jj++)
283         {
284             for(int kk=N_S_Y;kk<=N_MAX_Y-3;kk++)
285             {

```

```

286
287 T BETA=1+(KH-2)*log(fabs(v0[RN(jj,kk)
    ]/P_OP));
288 int KP=0;
289 {
290     const T PT=v0[RN(jj,kk)];
291
292     if (PT<v0[RN(N_LC-2,0)])
293     {
294         if (PT<v0[RN(N_LC-2,
            N_MAX_Y-3)])
295         {
296             HHP[jj][kk]=
                KH*KH*0.5*
                v0[RN(jj,
                    kk)]*(BETA
                        *v0[RN(jj,
                            kk)]/P_OP
                            -1)*(BETA*
                                v0[RN(jj,
                                    kk)]/P_OP
                                    -2);
297         }
298         else
299         {
300             for (;KP<
                N_MAX_Y
                -4&&PT<v0[
                    RN(N_LC-2,
                        KP)];KP++)
301                 ;
302             const T Q1=PT
                -v0[RN(
                    N_LC-1,KP
                    +1)];
303             const T Q2=PT
                -v0[RN(
                    N_LC-1,KP)
                ];
304             HHP[jj][kk]=(
                Q1*HHP_L[
                    KP]-Q2*
                    HHP_L[KP
                    +1])/(Q1-
                    Q2);
305         }
306     }
307     else
308     {
309         HHP[jj][kk]=0;
310     }

```

```

311         }
312         if (isnan (HHP [jj] [kk])) {HHP [jj] [kk]
           ]=0;}
313     }
314 }
315
316 //for NR_S<r<rLC
317 for (int jj=N_S_X; jj<=N_LC-2; jj++)
318 {
319     for (int kk=0; kk<=N_MAX_Y-3; kk++)
320     {
321         T BETA=1+(KH-2)*log (fabs (v0 [RN (jj, kk)
           ]/P_OP));
322         int KP=0;
323         {
324             const T PT=v0 [RN (jj, kk)];
325
326             if (PT<v0 [RN (N_LC-2, 0)])
327             {
328                 if (PT<v0 [RN (N_LC-2,
           N_MAX_Y-3)])
329                 {
330                     HHP [jj] [kk]=
                       KH*KH*0.5*
                       v0 [RN (jj,
                       kk)]*(BETA
                       *v0 [RN (jj,
                       kk)]/P_OP
                       -1)*(BETA*
                       v0 [RN (jj,
                       kk)]/P_OP
                       -2);
331                 }
332                 else
333                 {
334                     for (; KP<
                       N_MAX_Y
                       -4&&PT<v0 [
                       RN (N_LC-2,
                       KP)] ; KP++)
335                         ;
336                     const T Q1=PT
                       -v0 [RN (
                       N_LC-1, KP
                       +1)];
337                     const T Q2=PT
                       -v0 [RN (
                       N_LC-1, KP)
                       ];
338                     HHP [jj] [kk]=(
                       Q1*HHP_L[

```

```

339                                     KP]-Q2*
340                                     HHP_L[KP
341                                     +1)]/(Q1-
342                                     Q2);
343                                     }
344                                     }
345                                     }
346                                     }
347                                     }
348                                     }
349                                     }
350                                     }
351                                     }
352                                     }
353                                     }
354                                     }
355                                     }
356                                     }
357                                     }
358                                     }
359                                     }
360                                     }
361                                     }
362                                     }
363                                     }
364                                     }
365                                     }
366                                     }
367                                     }
368                                     }
369                                     }
370                                     }
371                                     }
372                                     }
373                                     }
374                                     }
375                                     }
376                                     }
377                                     }
378                                     }
379                                     }
380                                     }
381                                     }

```

```

382         T part1,part2;
383         if (x==N_MAX_X-3)
384         {
385             part1=(HHP[x][y]-HHP[x-1][y])
                    * (v0[RN(x,y)]-v0[RN(x-1,y)
                    ]) / (DX*DX);
386         }
387         else
388         {
389             part1=(HHP[x+1][y]-HHP[x][y])
                    * (v0[RN(x+1,y)]-v0[RN(x,y)
                    ]) / (DX*DX);
390         }
391         if (y==N_MAX_Y-3)
392         {
393             part2=(HHP[x][y]-HHP[x][y-1])
                    * (v0[RN(x,y)]-v0[RN(x,y-1)
                    ]) / (DY*DY);
394         }
395         else
396         {
397             part2=(HHP[x][y+1]-HHP[x][y])
                    * (v0[RN(x,y+1)]-v0[RN(x,y)
                    ]) / (DY*DY);
398         }
399         HHP_PRIME[x][y]=part1+part2;
400     }
401 }
402 }
403 }
404
405 //This represents the part of HHP that is a function of "R" and "Z" (
    NOT "PSI"). This includes any possible constant term.
406 __forceinline T f(int m, int n)
407 {
408     return 0;
409 }
410
411 #undef i
412 #undef j

```

BIBLIOGRAPHY

- [1] Scharlemann, E. T., Wagoner, R. V. 1973, *ApJ*, 182, 951
- [2] Contopoulos, I., Kazanas, D., Fendt, C. 1999, *ApJ*, 511, 351
- [3] Takamori, Y., Okawa, H., Takamoto, M., Suwa, Y. 2014, *PASJ*, 66, 25
- [4] Lovelace, R. V. E., Turner, L., Romanova, M. M. 2006, *ApJ*, 652, 1494
- [5] Michel, F. C. 1973, *ApJ*, 180, L133
- [6] Ogura J., Kojima Y. 2003, *Prog. Theor. Phys.*, 109, 619
- [7] Contopoulos, I., Kalapotharakos, C., Kazanas, D. 2014, *ApJ*, 781, 46
- [8] Petrova, S. A. 2013, *ApJ*, 764, 129